Broadview®
www.broadview.com.cn

Docker是改变世界的盒子，微服务的基石，
带领云计算进入2.0时代

Docker 容器实战

原理、架构与应用

廖煜　晏东　编著

快速入门：透过简单的理论讲解，带你进入 Docker 的世界
权威作者：笔者具有十多年虚拟化研究经验
步骤详细：手把手教你配置方法，为你量身定制自己的 Docker
内容丰富：揭露镜像制作过程，教你搭建镜像仓库

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

廖煜，电子科技大学计算机硕士，Ghostcloud 联合创始人。从事虚拟化技术研究十多年，拥有丰富的虚拟化、云计算及存储技术经验。国内第一批研究 Docker 容器技术的专家，Docker 开源项目贡献者。先后供职于 Intel、Symantec、EMC 等 IT 公司。从 2006 年开始，在 Intel MCP 实验室研究虚拟化技术。在 Symantec 期间，作为核心成员研发了 Symantec 第一款虚拟化产品 VxVI；并负责研发 Symantec 第一款存储一体机 N8000 系列。在 EMC 期间，主要从事 VNX 系列产品的研发测试工作。

晏东，Ghostcloud 创始人，超过 20 年编程经验，熟悉多种编程语言，全栈工程师。国内最早一批 Go 语言使用者，Docker 项目 Committer，Beego 项目 Committer，阿里云社区 Docker 技术专家。曾任索贝数码分布式文件系统及高可用中间件资深架构师，曾任 Symantec/Veritas 技术负责人，拥有超过 12 年分布式系统行业经验。

Docker 容器实战

原理、架构与应用

廖煜　晏东　编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以 Docker 实战为原则，通过各种应用实例详细介绍 Docker 基本原理、Docker 容器管理、Docker 镜像制作、Docker 仓库搭建等内容。本书注重 Docker 在不同场景的具体应用，专注于实用性和操作性。

本书共 14 章。涵盖的主要内容包括云计算简介、Docker 的安装、使用 Docker、Docker 深入解析、容器的网络、容器的数据、镜像仓库、镜像和容器的存储结构、定制 Docker Daemon、如何编写 Dockerfile、Dockerfile 最佳实践、使用容器提供服务、建立私有镜像仓库、Docker 常见问题等。

本书内容丰富，实例典型，实用性强。适合学习 Docker 的初学者、使用 Docker 的开发者及系统运维人员，尤其是需要在生产环境定制 Docker 的开发者和运维人员。

图书在版编目（CIP）数据

Docker 容器实战 ：原理、架构与应用/廖煜，晏东编著. —北京：电子工业出版社，2016.11
ISBN 978-7-121-30244-2

Ⅰ. ①D… Ⅱ. ①廖… ②晏… Ⅲ. ①Linux 操作系统—程序设计 Ⅳ. ①TP316.85

中国版本图书馆 CIP 数据核字（2016）第 260100 号

策划编辑：张月萍
责任编辑：张　慧
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编　100036
开　　本：787×1092　1/16　印张：17.25　　字数：441.6 千字
版　　次：2016 年 11 月第 1 版
印　　次：2016 年 11 月第 1 次印刷
定　　价：55.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn

前　言

为什么要写这本书?

在 2013 年 3 月，Docker 项目正式开源。短短的三年中，Docker 已经迅速普及开来，云计算、大数据、互联网等相关 IT 技术公司纷纷开始拥抱 Docker。在硅谷，有 200 多家 Docker 相关的创业公司。Google、Microsoft、AWS、IBM 等大型技术公司都已经加入 Docker 生态圈，开始使用 Docker，并为 Docker 社区共享。OpenStack、Hadoop 等云计算、大数据框架也开始向 Docker 迁移。

在国内，从 Docker 诞生之日起，各大技术公司和极客们就开始紧密关注这项技术。从 2014 年下半年开始，陆续有公司开始把现有系统迁移到 Docker 平台。BAT、华为、新浪、京东都有 Docker 相关产品上线。Docker 的普及愈演愈烈，大有掀起第二次云计算革命之势。

笔者从 2014 年年初开始接触 Docker，一下就被 Docker 的轻量性、便捷性所吸引。通过在实际项目中使用 Docker，发现 Docker 天生就是要解决敏捷开发、持续集成、持续发布、动态迁移、动态伸缩等互联网、云计算、大数据行业普遍存在的问题。通过把产品容器化，加速了开发、测试、发布的流程，产品发布时间从半天减少到 47 秒。通过镜像提交产品，解决了开发、测试、发布的环境异构性问题，使产品可以平滑地在各个部门之间传递。

在 Docker 的实际使用中，笔者遇到了很多棘手的问题，花费了大量的时间研究、分析、测试、解决这些问题。同时，笔者也发现很多初学者正在重复笔者走过的一些弯路。究其原因，是目前国内没有一本详细介绍 Docker 实战的书籍。因此，笔者决定把自己的一些经验总结出来，编撰成书，为广大读者服务。

本书有何特色?

1. 配置详细

本书涵盖 Docker Daemon、Docker 存储驱动、Docker 镜像仓库的所有配置选项，并对每个选项都有详细的介绍。

2. 注重实践性

本书从实践出发，介绍在实际应用场景中应该如何定制 Docker。详细介绍镜像制作的步骤、指令和最佳实践，各种存储驱动的区别和使用场合，以及 Docker Daemon

各种扩展功能和接口的使用方法，并列举了典型镜像的使用方法。

3．对 Docker 框架和原理进行分析

本书深入浅出地介绍 Docker 使用的核心技术：Namespace、CGroups 和 UnionFS。方便读者理解 Docker 原理，并在实际应用中可以更好地使用 Docker。

4．项目案例典型，实战性强，有较高的应用价值

本书中的第 11 章和第 12 章专门从实践出发，详细介绍镜像和容器的使用，并列出详细步骤，方便读者快速上手。

5．提供完善的技术支持和售后服务

本书提供专门的技术支持邮箱：book@ghostcloud.cn。读者在阅读本书过程中有任何疑问都可以通过该邮箱获得帮助。

本书内容及知识体系

第 1 篇　Docker 基础知识介绍（第 1～3 章）

本篇介绍云计算的历史和基本概念，Docker 的安装和基本使用。帮助读者对 Docker 有一个基本的了解，并搭建自己的 Docker 环境。

第 2 篇　Docker 的基本使用（第 4～7 章）

本篇介绍 Docker 的构架、Docker 的工作方式、下载镜像、制作镜像、运行容器、配置容器网络、在容器中实现数据持久化、备份还原迁移容器卷、关联容器代码做持续集成、查找镜像、下载镜像、上传镜像等内容。

第 3 篇　Docker 的高级使用（第 8～13 章）

本篇主要介绍 Docker 中的存储驱动、配置 Docker Daemon、制作镜像和搭建仓库等内容。

Docker 镜像提供了丰富的应用，对于 Docker 的流行起了重要作用。本篇详细介绍如何制作镜像，为读者介绍基本指令和最佳实践。Docker 镜像和容器有非常紧密的联系，本篇详细介绍两者的关系。

镜像和容器是通过 Docker 的存储驱动管理的。Docker 中有多种存储驱动，每种存储驱动在性能、可扩展性、安全性上有差别，不同应用场景应该选择不同的存储驱动。本篇详细介绍各种存储驱动，为读者在实际应用中选择存储驱动做指导。

Docker Daemon 是 Docker 管理镜像和容器的核心，除基本功能外，还提供很多扩展功能和接口。本篇详细介绍每种扩展功能和接口的具体使用方法。

第 4 篇　Docker 常见问题（第 14 章）

本篇主要总结 Docker 学习中遇到的一些问题，为读者提供统一的解释。

适合阅读本书的读者

- 在校计算机相关专业的学生；
- Docker 初学者；
- 基于 UNIX/Linux 环境的系统运维人员；
- 基于 UNIX/Linux 环境的测试人员；
- 基于 UNIX/Linux 环境的开发人员；
- 系统构架师；
- CTO；
- 互联网行业的开发、测试、运维人员；
- 初创公司的技术人员；
- 云计算、大数据行业的技术人员。

阅读本书的建议

- 没有云计算背景知识的读者，建议从第 1 章顺次阅读。
- 对于还没有使用过 Docker 的读者，建议从第 2 章开始阅读，首先搭建自己的实验环境。
- 对于特别关注 Docker 在存储方面的读写速度、稳定性、安全性的读者，建议详细阅读第 8 章。
- 对于需要在特定环境下定制 Docker Daemon 的读者，建议详细阅读第 9 章，学习如何修改 Docker Daemon 的配置，以适应具体应用场景。
- 对于希望在开发、测试、部署中使用 Docker 镜像提交产品的开发人员和运维人员，建议详细阅读第 10 章和第 11 章，学习如何制作镜像。
- 对于希望直接使用容器提供服务的读者，可以详细阅读第 12 章，学习如何使用官方镜像运行容器。
- 对于需要建立私有仓库管理镜像的读者，可以详细阅读第 13 章，学习搭建私有镜像仓库。
- 每章中都介绍详细的配置选项，读者需要通过实验，深刻理解和熟练地使用这些选项。
- 读者可以首先通读一遍本书，对 Docker 使用过程有一个大概的了解，然后根据自己的应用场景，详细阅读相关章节。

目　录

第 1 篇

Docker 基础知识介绍

第1章　云计算简介

云计算刚刚问世的时候，很多人都认为它的好处仅限于节约成本。但很快，大家就开始认识到云计算的强大，以及它对整个 IT 行业产生的深远影响。在云计算的推动下，IT 行业发生了深刻的变革。云计算将基础设施作为一种动态的、可自适应的资源提供给 IT 企业，一举解决了困扰业界许久的灵活性和响应性问题。随着云计算的兴起，像“cloud-native”和“cattle not pets”之类的术语行话也纷纷出现，它们都表达了一个意思，那就是整个和云计算相关的 IT 领域都需要彻底改变现有的意识形态，不能再像过去一样看待基础设施组件。因为在云计算模式下，基础设施已经不再是一个既庞大又昂贵，而且专业得令人难以企及的怪物，它不再需要烦琐的手动维护，也不再难以改换。

如果说云计算导致了 IT 业的变革，那么容器技术的出现则将这场变革提升到了新的高度。作为一种容器技术，Docker 已经以迅雷不及掩耳之势，迅速占领了业界的想象空间。容器一开始出现的时候遇到的情况和云计算差不多，大家都以为容器技术只不过是针对当前存在的封装和部署方面的问题提供了一个更便捷的解决方案而已。但实际上，容器虚拟化技术并不仅是一项更好的解决方案，它给我们的思维模式带来的变革要比云计算更为深远。

云计算改变了我们对“机器”的管理模式，但并未从本质上改变我们管理的对象。但容器就不一样，其优越性远超传统技术。借助容器技术，用户就能真正摆脱对服务器和操作系统的依赖，从而专注于应用及其组件本身。甚至可以这么说，容器技术和软件微服务架构代表了面向对象的组件式应用开发思想，同时也为云计算 2.0 提供了最核心的基础组件。本章主要从以下几个方面进行介绍。

- 虚拟化技术的分类和历史
- 云计算的几种服务模式
- Docker 的简单介绍
- Linux 快速入门
- 公有云主机快速入门

1.1　虚拟化技术的分类和历史

虚拟化一般分为硬件级虚拟化（hardware-level-virtualization）和操作系统级虚拟化（os-level-virtualization）。硬件级虚拟化是运行在硬件之上的虚拟化技术，它的管理软件也就是我们通常说的 hypervisor 或者 virtual machine monitor，它需要模拟的就是一个完整的操作系统，也就是我们通常所说的基于 Hyper-V 的虚拟化技术，VMWare、Xen、VirtualBox、AWS EC2 和阿里云 ECS 都是用的这种技术。操作系统级虚拟化是运行在操作系统之上的，它模拟的是运行在操作系统上的多个不同进程，并将其封装在一个密闭的容器里面，也称为容器化技术。Docker 正是容器虚拟化中目前最流行的一种实现。

1.1.1　硬件级虚拟化历史

硬件级虚拟化的历史非常久远，距今已超过半个世纪，但是在 VMWare、Hyper-V 及公有云技术兴起之前，在国内外都不是一个受人关注的行业。时至今日，当我们翻开这段历史的时候，不禁感叹美国在科技领域的前瞻性和领先地位。下面介绍一部分硬件级虚拟化的历史。

- 19 世纪 60 年代：美国出现了第一个虚拟化系统，它是由 IBM 开发的 CP-40 Mainframes 系统，虽然这个系统只是在实验室使用，但却为后来的 CP-67 系统奠定了基础。在那个时代，虚拟化系统主要由通用、贝尔实验室和 IBM 主导研发。
- 1987 年：一个非常强大的公司 Insignia Solutions 演示了一个称为 SoftPC 的软件模拟器，这个模拟器允许用户在 Unix Workstations 上运行 DOS 应用。在此之前这是不可能办到的。当时，一个可以运行 MS DOS 的个人计算机需要 1500 美元，而通过 SoftPC 模拟后，可降低到 500 美元。可以看出，当时的需求就是在大型工作站上运行微软的 DOS。到 1989 年，Insignia Solutions 发布了 Mac 版的 SoftPC，使苹果用户不仅能够运行 DOS，还能够运行 Windows 操作系统。
- 1997 年：随着 SoftPC 的一炮而红，其他虚拟化公司如雨后春笋般地出现了。在 1997 年，苹果开发了 Virtual PC，后来卖给了 Connectix。
- 1998 年：真正的王者 VMWare 出现了，他们在 1999 年开始销售 VMWare workstation，也就是我们很多人使用过的桌面版的虚拟机。
- 2001 年：VMWare 又发行了 ESX 和 GSX，也就是我们现在经常使用的 ESX-i 的前身。
- 2003 年：之前所说的 Connectix 被微软收购，后续推出了 Microsoft Virual PC，

再之后就没什么音讯了。同年，VMWare 也被 EMC 收购，成为 EMC 迄今最成功的一笔收购。就在这一年，一个开源的虚拟化项目 Xen 启动了，并在 2007 年被 Citrix 收购。

注意 Insignia Solutions 的衰败，Connectix 的没落，以及 VMWare 的半路杀出，都说明了商业和科技的竞争，正如一场马拉松，绝非一朝一夕，只有不断进步，才不会被淘汰。

1.1.2　操作系统级虚拟化历史

操作系统级虚拟化历史比硬件级虚拟化历史要短一些，尽管如此，其仍然有超过 30 年的历史，如果你是做 UNIX/Linux 开发的工程师，或许你已经在以往的工作中使用了这项技术，只是你不知道而已。

- 1982 年：你一定会很惊讶，第一个操作系统级的虚拟化技术是什么。答案就是 chroot，直到现在我们依然在使用的一个系统调用。这个系统调用会改变运行进程的工作目录，并且只能在这个目录里面工作。这种操作其实就是一种文件系统层的隔离。
- 2000 年：FreeBSD jail，真正意义上的第一个功能完整的操作系统级虚拟化技术。所以，真正的容器化技术从出现到现在已经过去了 16 年，并不是几年的时间。
- 2005 年：OpenVZ，这是 Linux 平台上的容器化技术实现，同时也是 LXC，即 Docker 最初使用的容器技术核心实现。
- 2008 年：LXC 发布，这是 Docker 最初使用的具体内核功能实现。
- 2013 年：Docker 发布，可以看出，Docker 本身是使用了 LXC，同时封装了其他的一些功能。Docker 的成功，与其说是技术的创新，还不如说是一次组合式的创新。

备注 曾经听一位资深人士说过，iPhone 你要说有多创新，真的说不上。手机很早就有了，电脑很早就有，触摸屏很早就有，但是苹果将所有这些有机地组合到了一起，再提供极致的用户体验，就产生了跨时代的产品。同样 Docker 所使用的技术也都不是新技术，它将这一系列技术有机地组合到一起，并提供极致的用户体验，就产生了跨时代意义的产品。

1.2　云计算服务模式

我们知道传统的服务器或者计算机主机，基本都是一锤子买卖，商家卖给你之后

基本就很难再从消费者身上获得其他收入。随着云概念的出现，越来越多的厂商都意识到卖硬件是不可能获得长期利益的，只有服务才是可持续的赢利点。因此，在 2010 年前后，出现了大批提供云服务的公司。总体来说，基本都可以归为下面几大类的一种或多种。

基础设施即服务（Infrastructure as a service，IaaS），通常指的是在云端为用户提供基础设施，如虚拟机、服务器、存储、负载均衡和网络等。亚马逊的 AWS 就是这个领域的佼佼者，国内则以阿里云为首。

平台即服务（Platform as a service，PaaS），通常指的是在云端为用户提供可执行环境、数据库、网站服务器和开发工具等。国外的 OpenShift、Red Hat、Cloudera、Cloud Foundry、Google App Engine 都是这个领域的公司，当然还有一个非常有名的公司，那就是 dotCloud，本书后续会再介绍一下这个公司。

软件即服务（Software as a service，SaaS），通常指的是在云端为用户提供软件，如 CRM 系统、邮件系统、在线协作和在线办公等。例如，微软就把自己的 Office 搬到了云端，国内的有道、麦客、Tower 都属于这个领域。

容器即服务（Container as a service，CaaS），随着容器的出现，在传统 IaaS 层出现了用容器替代虚拟机的服务模式，这种模式是虚拟云主机的升级版，由于容器的轻量级特性，从资源利用率和性能方面都比 IaaS 层的虚拟机高出很多。

一般认为 IaaS、PaaS 和 SaaS 是云计算最基本的三种服务模式，其分层结构如图 1.1 所示。

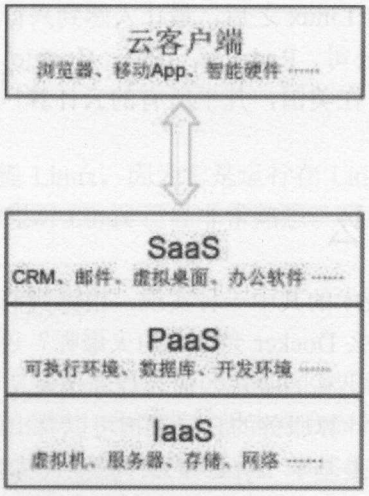


图 1.1　云计算服务模式图

1.3　Docker 介绍

了解了虚拟化的基础知识后，可以知道 Docker 是容器化技术的一种，那它到底有

什么特别之处呢？

1.3.1　Docker 主要解决什么问题

Docker 对外宣称的是 Build、Ship 和 Run，Docker 要解决的核心问题就是快速地干这三件事情。它通过将运行环境和应用程序打包到一起，来解决部署的环境依赖问题，真正做到跨平台的分发和使用。而这一点和 DevOps 不谋而合，通过 Docker 可以大大提升开发、测试和运维的效率。在这个移动互联网的时代，如果一个工具能够节省人力，提升效率，必定会流行起来。

1.3.2　Docker 的历史

之前提到一家公司叫 DotCloud，这家公司是一家法国的公司，最初也是提供 PaaS 服务的，他们提供了对多种语言的运行环境支持，如 Java、Python、Ruby、Node.js 等。但是，可能叫生不逢时吧，在 PaaS 领域有太多的巨头和大企业了，有一天 Solomon Hykes（Docker 之父）就召集了公司的哥儿几个来商量了一下，最后得出的结论是，如果要和那些大厂商硬干肯定是不行的，那么干脆就把他们做的项目 Docker 开源了。即使赚不到钱，至少也在开源社区得到一个好名声。

因此，在 2013 年 3 月，Docker 正式以开源软件形式发布了。正是由于这次开源，让容器领域焕发了第二春，截至 2015 年 11 月，Docker 在 GitHub 上收到了 25600 个赞，超过 6800 次克隆，以及超过 1100 名的贡献者，成为 20 个最具影响力的 GitHub 开源项目。可以说，Docker 是继 Linux 之后，最让人感到兴奋的系统层面的开源项目。据不完全统计，包括 Docker 公司、Red Hat、IBM、Google、Cisco、亚马逊及国内的华为等，都在为它贡献代码。在美国，几乎所有的云计算厂商都在拥抱 Docker 这个生态圈。

1.3.3　Docker 是什么

Docker 其实是容器化技术的其中一种实现，根据我们之前的介绍，容器化技术并不是最近才出现的，那为什么 Docker 会如此的火爆呢？还是这个时代造就的，因为我们处在一个云计算发展异常迅猛的时代，而云计算又是所有移动互联网、IT，以及未来消费者行业的基础。从云计算服务的三层架构可以看出，传统的 IaaS 层、虚拟机是基础组成部分，而虚拟机都是基于 Hyper-V 架构的，也就是说，每一个虚拟机都会运行一个完整的操作系统，一个操作系统至少需要占用 5GB 左右的磁盘空间，但是操作系统对于我们来说是完全无用的，我们关心的是虚拟主机所能提供的服务。因此，迫切需要轻量级的主机，那就是 Docker 容器。我们可以看一下，Docker 和虚拟机的区别（如图 1.2 所示）。

从图 1.2 可以看到，容器由于省去了操作系统，整个层级更简化，可以在单台服务器上运行更多的应用，而这正是 IaaS 所需要的，可能 5GB 左右的空间对你来说不是

什么大事，但是如果需要对外提供成千上万的主机，那就是不得不考虑的问题，而这正是容器虚拟化要解决的问题。

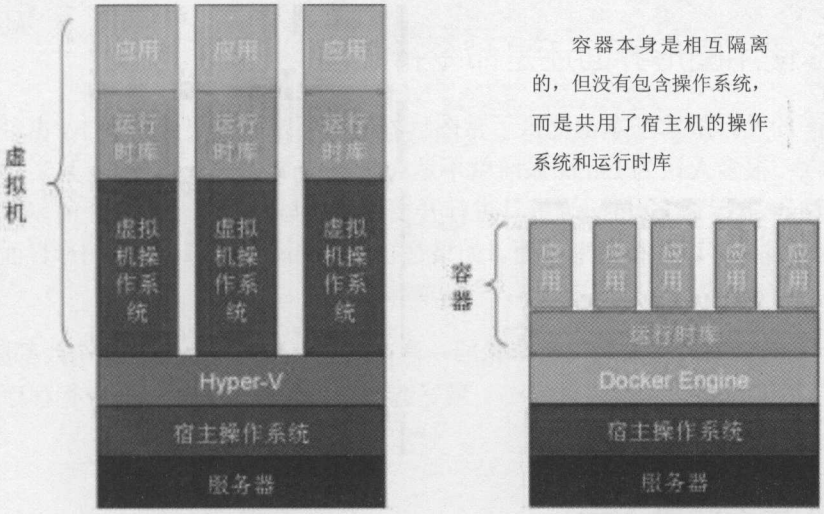


图 1.2　虚拟机 vs 容器

备注　每到这里，我都喜欢举一个例子。波音公司造飞机肯定不会考虑在水上航行的问题，造船厂也绝对不会考虑在天上飞的问题，汽车制造公司也不会考虑在水上跑的问题，那么对于广大的移动互联网公司和云计算公司，也可以只关注最顶层的应用，而不需要考虑操作系统的问题。

1.4　Linux 快速入门

讲解 Docker 就不得不提 Linux，因为它是运行在 Linux 下的，而且最基础的镜像也是 Linux 的发行版本，如果对 Linux 已经非常熟悉，可以跳过进入下一节。

1.4.1　选取什么发行版本

Linux 包含了很多的发行版本，包括 Ubuntu、CentOs、RedHat、Federa 等，但是它们都是基于 Linux kernel 的版本，各个发行版本都会做相应的包装、优化和简化，但是基本上内核版本不会有太大的差异。根据经验，本书推荐使用 Ubuntu 或者 CentOs。Ubuntu 的优点如下：

- 内核更新及时；
- 软件安装和更新方便；
- GUI 简单实用，CentOS 就是 Red Hat Enterprise 的开源版本，也是不错的选择，考虑到 Ubuntu 对 Docker 的完美支持，本书推荐使用 Ubuntu。Ubuntu 可以在

Docker 的存储部分使用 AUFS，而 CentOS 只能使用 Devicemapper，前者的性能要稍好一些。

1.4.2　使用图形界面还是命令行界面

Linux 的发行版本基本都提供了带图形界面和不带图形界面的版本，也可以在安装后进行调整。很多人认为 Linux 系统就不适合带 IDE 界面的开发环境，其实是错误的。可以在 Linux 下运行 Eclipse 等工具进行开发，而且与其他平台没有任何差异。所以如果要进行开发，可以选择图形界面。如果是初学 Linux，也可以使用图形界面，来进行过渡。

但是又回到本书之前的飞机—轮船—汽车理论，凡是没有必要的东西都应该去掉，所以如果使用 Linux 运行后台程序，最好选择服务器版，只运行命令行程序，多使用命令行也是一种学习和锻炼。

1.4.3　英文还是中文

一般来说，使用 Linux 的用户都是相对专业的用户，我建议一律使用英文操作系统。英文的系统可以强迫你学习英语，同时，系统不会出现乱码。目前，很多一手的技术资料基本都是英文的，英文好是非常大的优势。

1.4.4　安装 Ubuntu 14.04

（1）下载镜像 http://www.ubuntu.org.cn/download/server。

（2）通过 ISO 启动系统，并选择 English，如图 1.3 所示。

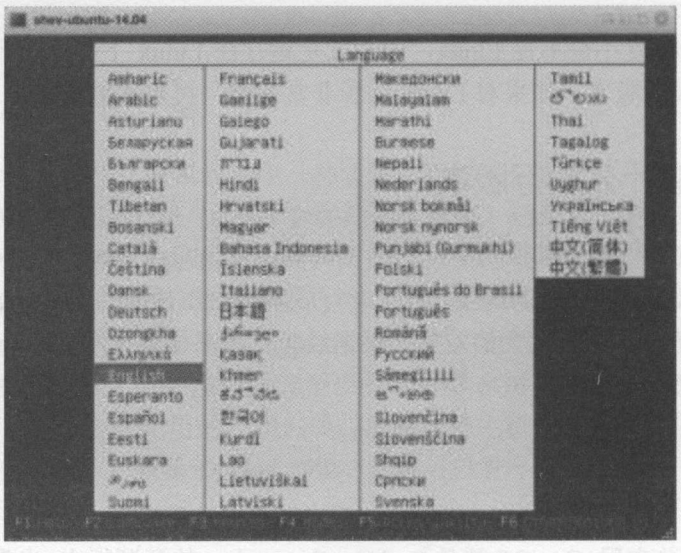


图 1.3　选择 English

（3）选择区域为 Hong Kong，之后都选用默认选项，如图 1.4 所示。

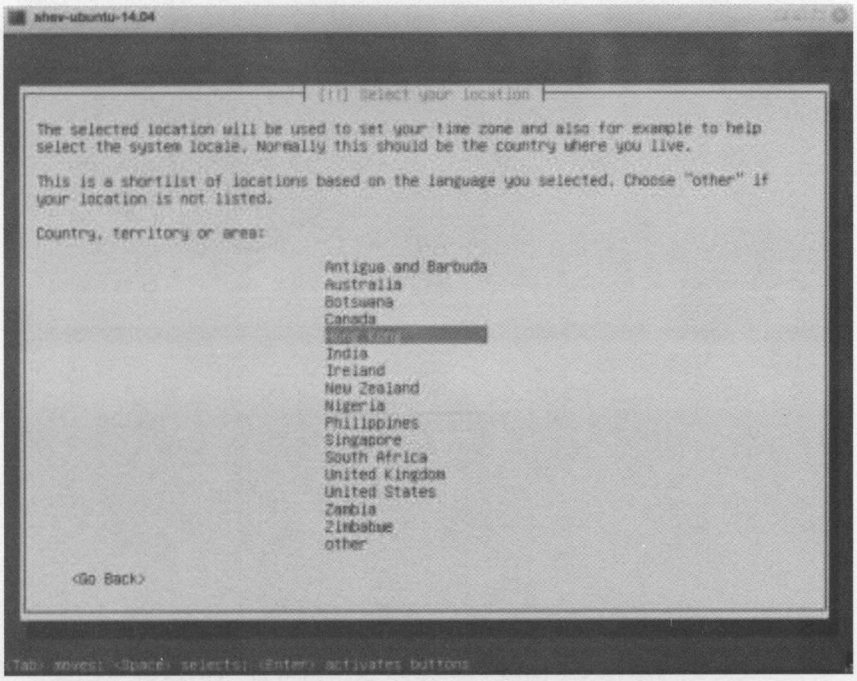


图 1.4　选择地区

（4）设置用户名和密码，如图 1.5 所示。

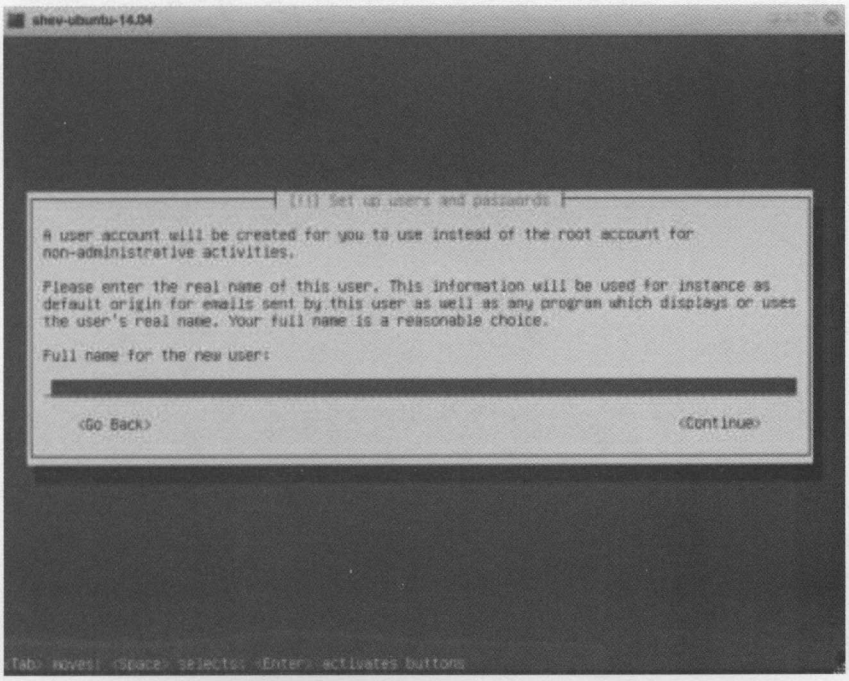


图 1.5　设置用户名和密码

（5）之后的设置都按默认选项即可，在划分磁盘时，根据说明进行操作。之后就

进入了安装阶段，如图 1.6 所示。

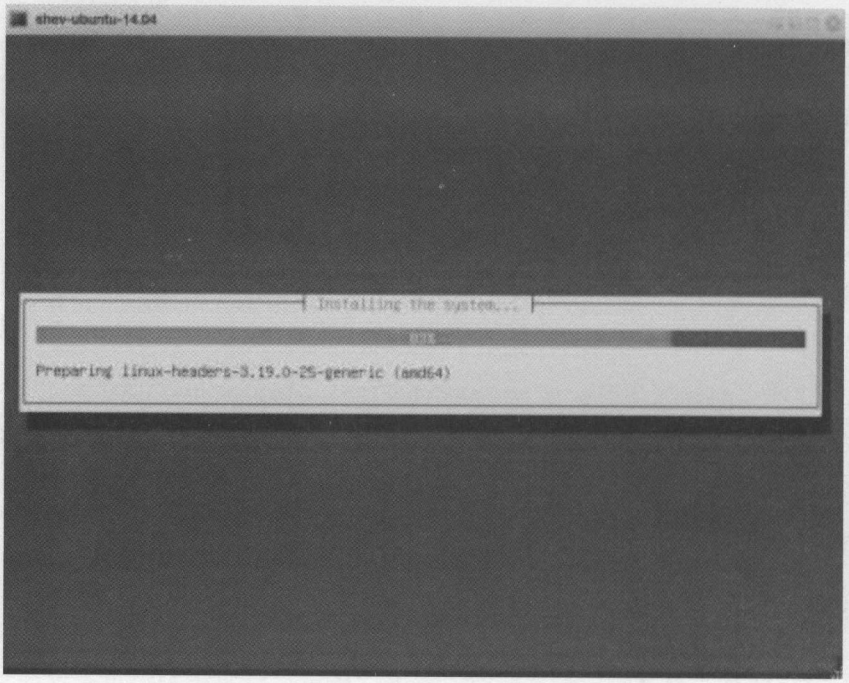


图 1.6　安装过程中

（6）在软件选择时，勾选 OpenSSH server，方便之后远程登录，如图 1.7 所示。

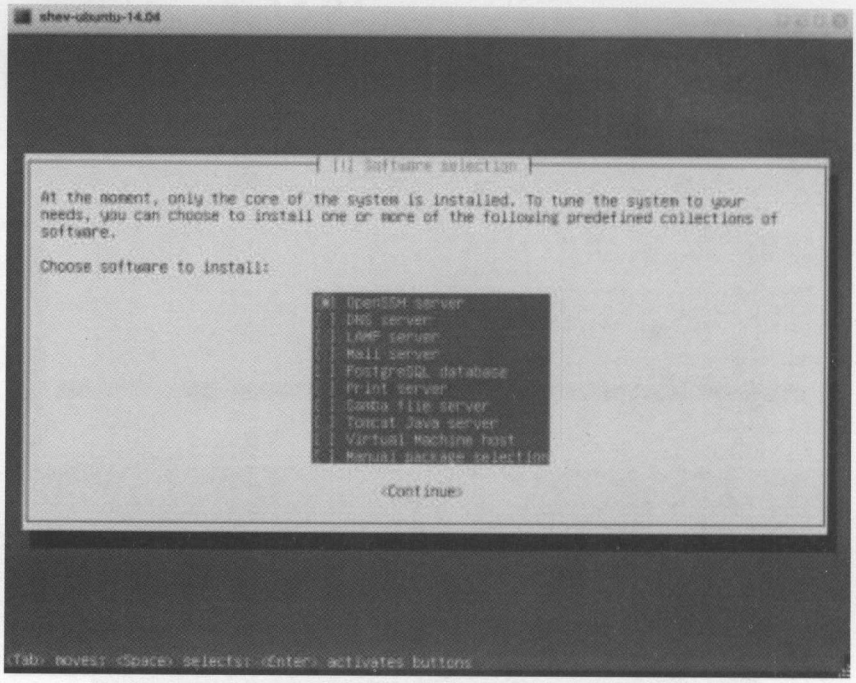


图 1.7　勾选 OpenSSH server

（7）安装结束，如图 1.8 所示。

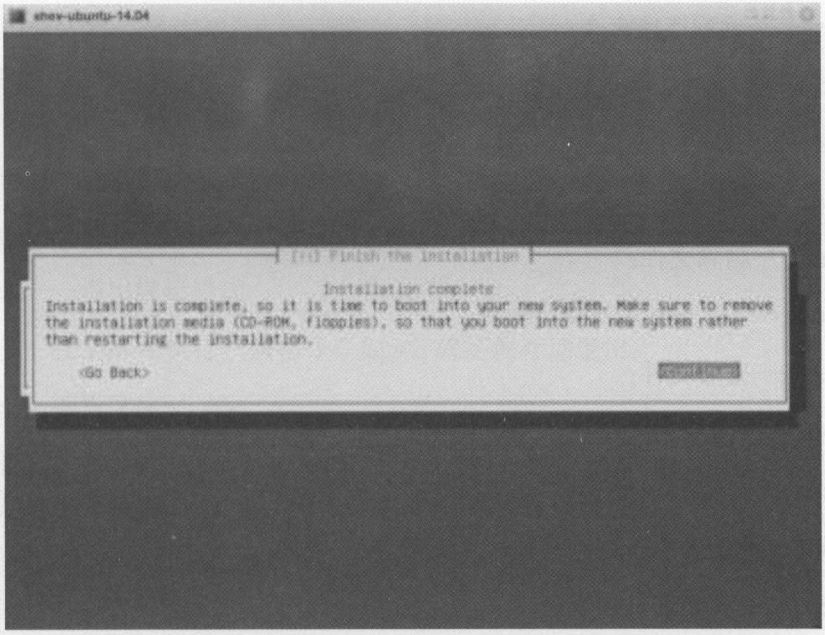


图 1.8　安装完毕

（8）重启后，输入用户名和密码进入系统，如图 1.9 所示。

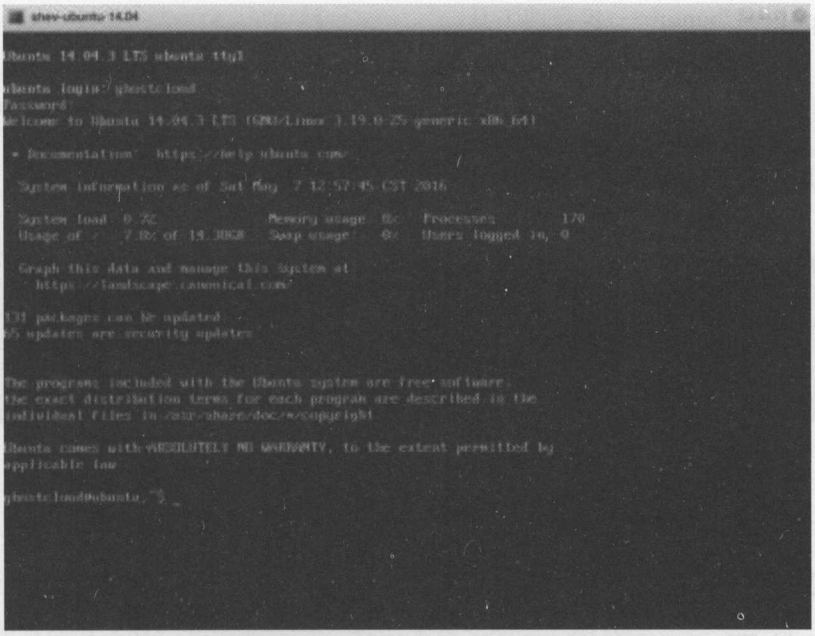


图 1.9　登录进入系统

1.4.5　Linux 常用工具

如果你完全没有 Linux 基础，需要至少熟悉以下工具或者命令：

- ssh——远程连接命令；
- vim——一个 Linux 下命令行编辑文件的工具；
- ls——列举文件及文件夹；
- cp——复制文件；
- rm——删除文件；
- sudo——以 root 用户执行命令；
- cat——查看文件；
- pwd——查看当前路径；
- mkdir——创建文件夹；
- find——查找文件；
- grep——搜索文件内容；
- which——查看命令在什么位置；
- tar——打包和压缩命令；
- apt-get-Ubuntu——包管理工具。

1.4.6　启用 root 用户

root 用户是 Linux 的最高权限用户，相当于 Windows 的超级管理员。可以通过下面的方式启用 root 用户，如图 1.10 所示。

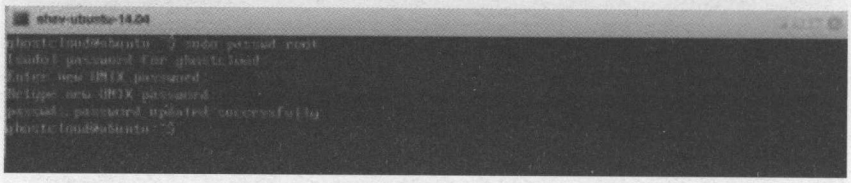


图 1.10　启用 root 用户

根据提示输入当前用户的密码，然后再输入 root 的密码。Sudo 表示以管理员身份运行命令。

1.4.7　使用 vim

vim 是 Ubuntu 默认的文本编辑器，学习使用 Linux 第一步就是学会使用 vi。有的时候 vim 可能是没有安装的，需要手动进行安装。

```
root@ghostcloud:~# sudo apt-get update && apt-get install vim
Hit http://mirrors.163.com precise Release.gpg
Hit http://mirrors.163.com precise Release
```

```
Hit http://apt.ghostcloud.cn ubuntu-precise Release.gpg
```

安装成功之后，就可以使用 vim 了。vim 是 vi 的升级版，有了很多优化。常用的命令有：

- i——从当前位置开始插入数据；
- a——在当前位置后面插入数据；
- esc——退出编辑模式；
- :——在 vim 中执行一条指令，如 wq 就是保存加退出；
- /——搜索文字；
- 上下左右键——移动光标，vi 里面不能用方向键，但是 vim 里面是可以使用的。虽然还有很多命令，但是使用上面的基本就能操作了。

1.4.8　配置网络

Ubuntu 的网络配置是放在/etc/network/interfaces 下的，通过 vim 进行查看和修改。

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).
# The loopback network interface
auto lo
iface lo inet loopback
# The primary network interface
auto eth0
iface eth0 inet static
address 192.168.1.10
netmask 255.255.0.0
gateway 192.168.0.1
dns-nameservers 61.139.2.69 218.6.200.139
```

修改完毕后，需要重启网络，一个比较好的方式是禁用再启用网络:

```
root@ghostcloud:~# ifdown -a && ifup -a
```

1.4.9　启用 SSH Server

SSH 是 Secure Shell 的缩写，是 Linux 的标准远程连接工具，通过这个工具可以以命令行的方式远程连接到 Linux 主机之上。首先需要检查在主机上是否安装了 SSH Server。如果没有安装，则需要在客户机上安装 openssh-server。

```
root@ghostcloud:~# dpkg -l | grep openssh-server
ii  openssh-server  1:6.6p1-2ubuntu2  amd64  ecure shell (SSH) server, for secure
access from remote machines
```

接下来，需要配置 sshd。

```
# vim /etc/ssh/sshd_config
#允许 root 登录
```

```
PermitRootLogin yes
#允许通过密码进行验证登录
PasswordAuthentication yes
```

保存退出后，执行。

```
root@ghostcloud:~# restart ssh
```

验证能否本地登录。

```
root@ghostcloud:~# ssh root@localhost
The authenticity of host 'localhost (::1)' can't be established.
ECDSA key fingerprint is 3a:8c:00:76:4d:4d:62:a7:c7:18:a0:00:e6:d0:17:c7.
Are you sure you want to continue connecting (yes/no)?
```

根据提示输入用户密码，如果可以登录说明安装成功，最后执行 exit，退出 SSH 连接。

1.4.10　通过客户端远程连接 Linux 主机

在客户机上需要安装 SSH Client，并通过 SSH Client 远程访问主机。

首先检查客户机上是否安装 SSH Client。如果没有安装，则需要在客户机上安装 openssh-client。

```
root@ghostcloud:~# dpkg -l | grep openssh-client
ii  openssh-client                    1:6.6p1-2ubuntu2                amd64
secure shell (SSH) client, for secure access to remote machines
```

获得远程主机的 IP、端口、用户名和密码，通过下面的命令登录远端主机。其中，-p 参数指定 SSH Server 的服务端口。根据提示输入用户密码，验证通过后即可以登录远程主机。

```
root@ghostcloud:~# ssh -p 22 root@192.168.0.1
```

第一次访问远程主机时，需要把远程主机加入客户端信任列表。根据提示，在终端输入 yes。

```
root@ghostcloud:~# ssh -p 22 root@192.168.0.1
The authenticity of host '[192.168.0.1]:22 ([192.168.0.1]:22)' can't be
established.
ECDSA key fingerprint is 08:1d:db:e4:d2:e0:87:89:ed:ca:69:82:17:6a:83:57.
Are you sure you want to continue connecting (yes/no)? yes
```

SSH Server 默认使用 22 端口，不加-p 参数时，SSH Client 默认连接远程主机的 22 端口。如果没有修改 SSH Server 的默认端口，则在 SSH Client 访问远程主机时，可以忽略-p 参数。

```
root@ghostcloud:~# ssh root@192.168.0.1
```

1.4.11　免密码登录 Linux 主机

免密码登录的原理是在需要登录的远程主机上，存放当前机器的公钥。

（1）在当前机器生成公钥和私钥 ssh-keygen。

（2）根据提示生成以后，会在~/.ssh/目录下生成相关的文件。这里的~指的是用户的目录。例如，在 Linux 下 abc 用户的目录为/home/abc，root 用户的目录为/root，在 Mac OS 下是在/Users/<用户名>。

（3）将公钥 id_rsa.pub 复制到目标机器上，scp ~/.ssh/id_rsa.pub root@192.168.1.10:~/ 这行命令将当前用户的公钥复制到远程机器的 root 用户目录下。

（4）ssh root@192.168.1.10。

（5）ssh-keygen #在远端产生密钥。

（6）cat id_rsa.pub >> ~/.ssh/authorized_keys #加入信任列表。

（7）rm id_rsa.pub #删除公钥。

（8）exit #退出远程机器。这时已经返回到当前机器，再执行 ssh root@192.168.1.10 就不再需要输入密码了。

1.4.12　安装软件

Ubuntu 软件安装使用的是和 Debian 一样的系统——apt。apt 就是一个软件仓库，只需指定仓库地址，就可以进行搜索和安装。

（1）添加源：默认的 Ubuntu 源是指向国外的，位于/etc/apt/sources.list，可以在网上搜索国内的源，如网易源，阿里源等。

```
deb http://mirrors.163.com/ubuntu/ trusty main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-security main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-updates main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-proposed main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-backports main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-security main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-updates main restricted universe multiverse
```

（2）搜索软件。

```
root@ghostcloud:/etc/apt# apt-cache search apache2 | grep ^apache2
apache2 - Apache HTTP Server
apache2-bin - Apache HTTP Server (binary files and modules)
apache2-data - Apache HTTP Server (common files)
apache2-dbg - Apache debugging symbols
```

```
apache2-dev - Apache HTTP Server (development headers)
apache2-doc - Apache HTTP Server (on-site documentation)
apache2-mpm-event - transitional event MPM package for apache2
apache2-mpm-prefork - transitional prefork MPM package for apache2
apache2-mpm-worker - transitional worker MPM package for apache2
apache2-utils - Apache HTTP Server (utility programs for web servers)
apache2.2-bin - Transitional package for apache2-bin
apache2-mpm-itk - transitional itk MPM package for apache2
apache2-suexec - transitional package for apache2-suexec-pristine
apache2-suexec-custom - Apache HTTP Server configurable suexec program for mod_suexec
apache2-suexec-pristine - Apache HTTP Server standard suexec program for mod_suexec
```

（3）安装软件。

```
root@ghostcloud:/etc/apt# apt-get install apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
apache2-bin apache2-data libapr1 libaprutil1 libaprutil1-dbd-sqlite3
libaprutil1-ldap ssl-cert
Suggested packages:
apache2-doc apache2-suexec-pristine apache2-suexec-custom apache2-utils
openssl-blacklist
The following NEW packages will be installed:
apache2 apache2-bin apache2-data libapr1 libaprutil1 libaprutil1-dbd-sqlite3
libaprutil1-ldap ssl-cert
0 upgraded, 8 newly installed, 0 to remove and 72 not upgraded.
Need to get 1285 kB of archives.
After this operation, 5348 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

（4）卸载软件。

```
root@ghostcloud:/etc/apt# apt-get purge apache2
```

（5）查找本地安装的软件。

```
root@ghostcloud:/etc/apt# dpkg -l
```

1.4.13　公有云主机快速入门

公有云主机，其实就是将主机托管到公网上，用户不需要关心如何管理主机，把这些事情交给专业的厂商。目前能提供公有云主机的厂商有很多，在国外有 Amazon AWS、Microsoft Azure、Google App Engine，在国内有阿里云、UCloud、青云等，接下来就以国内最常用的阿里云 ECS 为例讲解如何使用公有云主机。阿里云是国内目前最大的公有云提供商，它的产品线也比较广，不过最出名的还是 ECS 云主机。如果对公有云主机非常熟悉，可以跳过本节。

1.4.14　购买云主机

（1）首先打开主页 https://www.aliyun.com/ ，注册用户之后就进入官方页面，选择产品→云服务器 ECS，如图 1.11 所示。

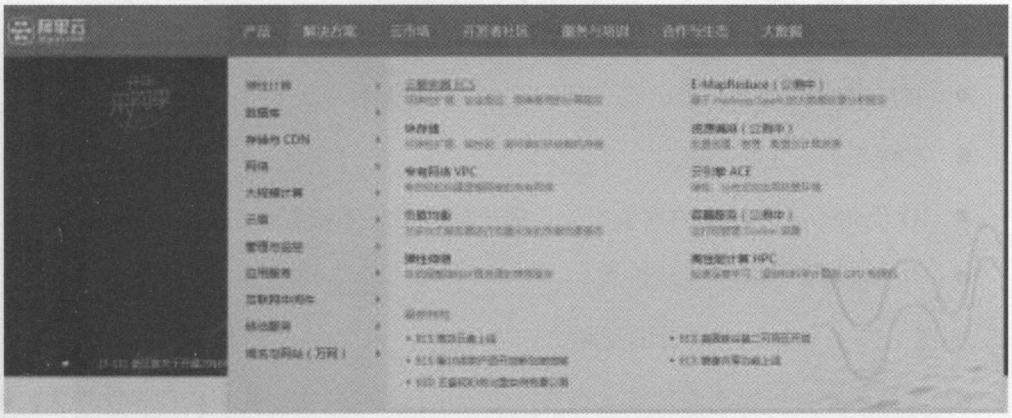


图 1.11　阿里云首页

（2）服务器选型。

阿里云的服务器是根据配置进行收费的，用户可以根据需求进行购买，如图 1.12 所示。

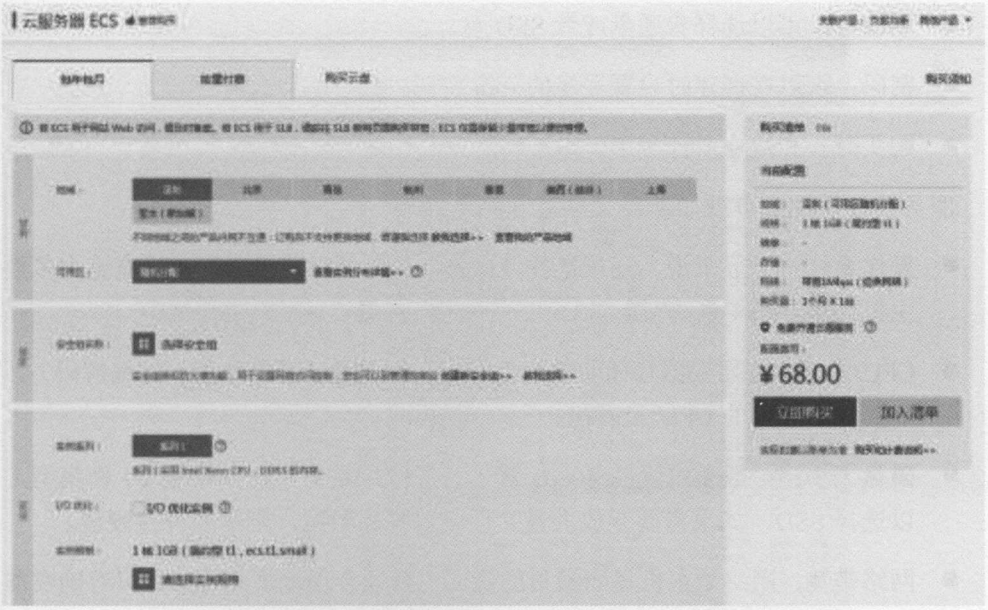


图 1.12　阿里云服务器选型 1

由于页面比较长，在这里分为两个部分进行截取。其中，

- 地域：购买的主机所在的区域，每个区域内的主机都拥有一个公网 IP 和一个

私网 IP。在同一个区域内，私网 IP 是可以直接访问的，一般是 10.*开头的 IP 地址。公网 IP 是可以在任何地方进行访问的，同时还以作为将网站的域名指向该 IP。阿里云的双网结构，给用户带来了很大的方便，其中，内网之间的速度是非常快的，可以充分利用这一特性，配合阿里云的其他相关产品使用，如对象存储。

- 可用区：可以理解为一个地域里面的不同接入点。
- 安全组：类似防火墙功能，用于设置网络访问控制，初期可以忽略。
- 实例系列：主机类型。
- IO 优化：可以加速的增值服务，但价格很贵，不推荐使用。
- 实例规格：具体的主机配置。

如图 1.13 所示，这一部分的配置方法如下。

- 公网带宽：可以按固定带宽收费，也可以按流量收费。如果访问量比较大，最好选择固定带宽收费。
- 带宽：公网带宽数值，价格很贵，慎重选择。
- 镜像类型：包括公共镜像、自定义镜像、共享镜像和镜像市场。在镜像市场中，可以购买一些第三方服务商的镜像，直接使用。
- 系统盘：可以选择普通盘或者 SSD 盘。
- 密码：可以在创建时设置登录的 root 密码。
- 购买时长和数量：计费用。

服务器选型注意事项如下。

- 操作系统：一般来说，建议选择 Linux，具有更好的灵活性，毕竟绝大多数云主机都是 Linux 系统。
- CPU/内存：需要根据具体业务合理配置，如果是计算密集型（如编解码），则需要选择更好的 CPU 和内存。
- 磁盘大小：一般系统盘的 40GB 对于普通应用足够了，如果是 IO 密集型，可以选择 SSD，如果需要存放大量文件或数据库数据，则需要外接硬盘。
- 网络带宽：网络是主机里面最贵的资源，必须合理地规划网络，只有确实需要大带宽时才购买，如果不对公网提供访问则尽量使用内部带宽，既省钱又快速。

本书选择 Ubuntu 14.04 作为主机镜像，点击购买、支付后就可以在控制台中看到相关的主机信息。

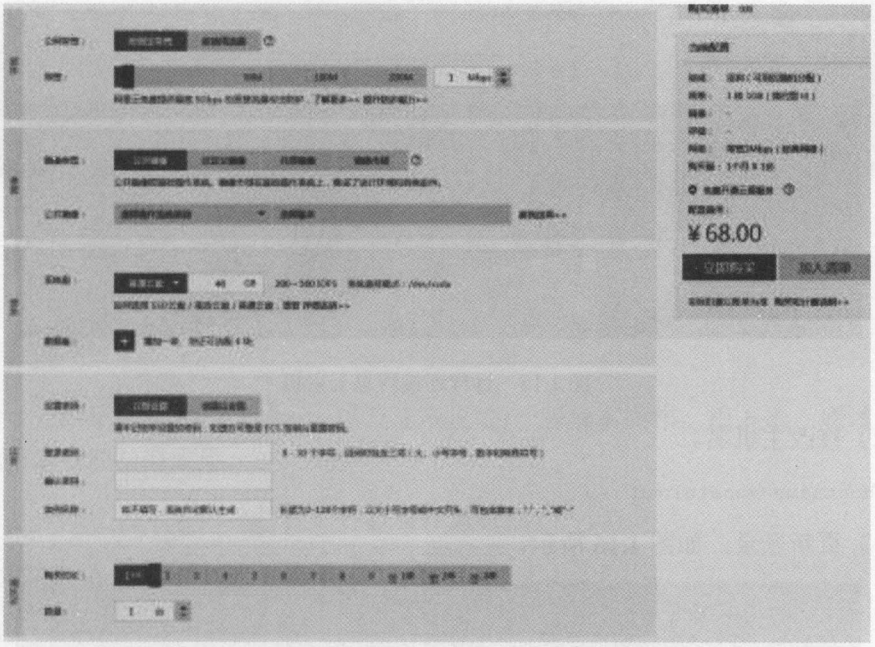


图 1.13　阿里云服务器选型 2

1.4.15　连接到云主机

（1）点击位于顶部的“管理控制台”，可以看到现在的主机列表，如图 1.14 所示。

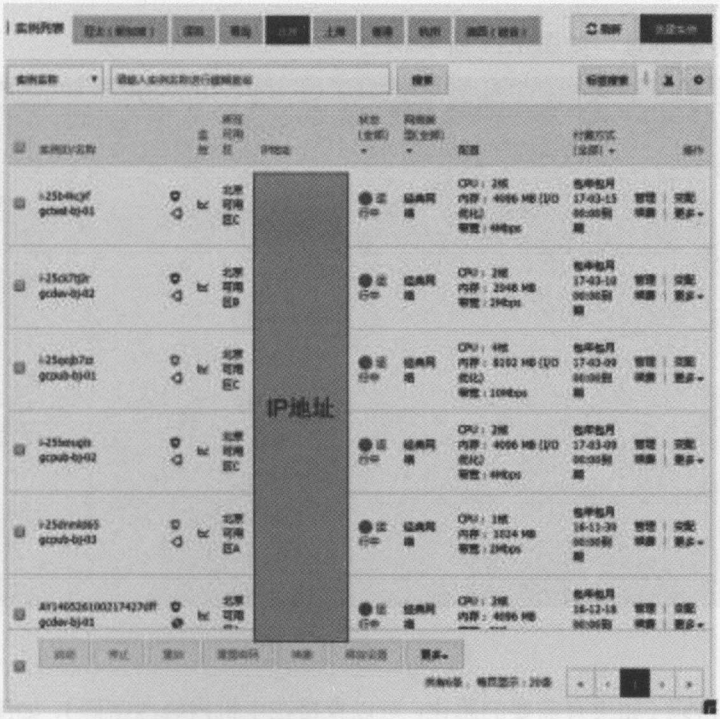


图 1.14　阿里云主机后台管理

新购买的机器状态可能是在启动中，图 1.14 所有主机都处于运行状态，在 IP 地址列可以看到公网 IP 和私网 IP，接下来将通过 SSH 进行远程连接，如图 1.15 所示。

```
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-65-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

Welcome to aliyun Elastic Compute Service!

Last login: Mon Mar 14 11:20:14 2016 from 182.148.56.118
root@iz25b4kcjrfz:~#
```

图 1.15　远程连接阿里云主机

（2）修改主机名。

```
# hostname ghostcloud
```

（3）重新登录，如图 1.16 所示。

```
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-65-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

Welcome to aliyun Elastic Compute Service!

Last login: Mon Mar 14 11:23:14 2016 from 182.148.56.118
root@ghostcloud:~#
```

图 1.16　修改主机名

（4）获取 IP 地址，如图 1.17 所示。

```
root@ghostcloud:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:16:3e:1c:01:3e
          inet addr:[redacted]  Bcast:10.46.183.255  Mask:255.255.248.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:701 errors:0 dropped:0 overruns:0 frame:0
          TX packets:43 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:51948 (51.9 KB)  TX bytes:6159 (6.1 KB)

eth1      Link encap:Ethernet  HWaddr 00:16:3e:1c:0c:a7
          inet addr:1[redacted]34  Bcast:101.201.31.255  Mask:255.255.252.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:20394 errors:0 dropped:0 overruns:0 frame:0
          TX packets:558 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1500968 (1.5 MB)  TX bytes:80097 (80.0 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:648 errors:0 dropped:0 overruns:0 frame:0
          TX packets:648 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:51924 (51.9 KB)  TX bytes:51924 (51.9 KB)

root@ghostcloud:~#
```

图 1.17　获取 IP 地址

（5）修改密码（如果有需要的恶化，可以直接在主机内修改密码），如图 1.18 所示。

```
root@ghostcloud:~# passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@ghostcloud:~#
```

图 1.18　修改主机密码

1.5　习题

本章介绍了云计算、Docker，以及 Linux 的一些基本操作。接下来，通过习题和实验检验本章的学习成果。

（1）虚拟化技术分为哪两种？各自有什么特点？

（2）Docker 与虚拟机有什么区别？

（3）什么是进程？

（4）结合本章知识，分析什么场景使用虚拟机，什么场景使用 Docker。

（5）简要描述 SSH Server 的公钥和私钥。

（6）购买一台 Linux 云主机，并通过 vim 修改默认的 DNS Server。

第2章　Docker的安装

Docker公司维护了多个和Docker相关的项目，每个项目的侧重点都不一样，本书所讲的Docker安装指的是Docker Engine的安装，即Docker最核心的容器处理部分。Docker原生态是只能运行在Linux系统下，目前几乎所有的Linux发行版本都可以运行Docker，包括Ubuntu、CentOS、RHEL、SuSE等。本章将学习以下知识：

- 在Ubuntu下安装Docker；
- 在CentOS下安装Docker；
- 通过第三方平台安装Docker；
- 在Mac OS下安装Docker；
- 在Windows下安装Docker。

2.1　在Ubuntu下安装Docker

Docker官方支持的Ubuntu版本包括：

- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

请注意，Ubuntu Utopic Unicorn 14.10和15.04并不是官方支持的版本。同时，由于12.04太过陈旧，也不建议使用。

2.1.1　前置条件

Docker需要使用Linux中内核的CGroups和Namespace功能，所以必须使用包含这两项功能的Linux内核。因此，Linux内核必须是高于3.10的64位版本，可以通过

uname -r 查看当前的内核版本。

```
root@ghostcloud:~# uname -r
3.13.0-65-generic
```

2.1.2　更新 apt 源

apt 是 Ubuntu 默认的包管理系统，apt 在安装时会根据实际 apt 配置文件搜索安装源。一个系统可以包含多个不同的安装源，安装时 apt 会逐个进行搜索，Docker 官方的 apt 仓库只包含 Docker-engine 的安装源，对于其依赖的包并不在内。因此，在设置 Docker 源前，需要针对国内的环境设置 apt 源。由于国内外网速的差距和 GFW 的原因，如果不设置 apt 源则安装时间会很长，同时可能安装失败。

（1）以 root 用户登录系统。

（2）更新国内 Ubuntu 源。在/etc/apt/sources.list.d/目录下，新建一个 ghostcloud.list 文件（可以是任意文件名），然后加入一行：

```
deb http://mirrors.163.com/ubuntu/ trusty main
```

（3）将新加的文件对应的仓库信息加到 apt 系统中，并安装 https 的 CA 证书。

```
root@ghostcloud:~# apt-get update
root@ghostcloud:~# apt-getinstallapt-transport-https ca-certificates
```

（4）添加 GPG key，这是访问 Docker 源的公钥。

```
root@ghostcloud:~# apt-key adv --keyserver
hkp://p80.pool.sks-keyservers.net: 80--recv-keys58118E89F3A912897C070ADBF7622
1572C52609D
```

（5）打开/etc/apt/sources.list.d/ghostcloud.list，根据系统加入 Docker 安装源。

Ubuntu Trusty 14.04(LTS)：

```
deb https://apt.dockerproject.org/repo ubuntu-trusty main
```

Ubuntu wily 15.10：

```
deb https://apt.dockerproject.org/repo ubuntu-wily main
```

（6）再次更新源。

```
root@ghostcloud:~# apt-get update
```

（7）删除旧的 LXC。

```
root@ghostcloud:~# apt-purge lxc-docker
```

（8）检查 apt 是否从 Docker 官方源安装。

```
root@ghostcloud:~# apt-cache policy docker-engine
docker-engine:
  Installed: 1.10.3-0~trusty
  Candidate: 1.10.3-0~trusty
```

```
Version table:
*** 1.10.3-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
        100 /var/lib/dpkg/status
     1.10.2-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.10.1-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.10.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.9.1-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.9.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
        500 http://apt.ghostcloud.cn/repo/ ubuntu-trusty/main amd64 Packages
     1.8.3-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.8.2-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.8.1-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.8.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.7.1-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.7.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.6.2-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.6.1-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.6.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     1.5.0-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

2.1.3　Ubuntu 14.04 特殊处理

对于 Ubuntu 14.04，官方推荐安装 linux-image-extra 内核包，这个包允许用户使用 AUFS，AUFS 可以在某种程度上提升容器的 IO 性能。通过以下语句进行安装。

```
root@ghostcloud:~# apt-get install linux-image-extra-$(uname -r)
Reading package lists... Done
Building dependency tree
Reading state information... Done
linux-image-extra-3.19.0-25-generic is already the newest version.
linux-image-extra-3.19.0-25-generic set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 126 not upgraded.
```

2.1.4　正式安装

同样，以 root 用户登录系统。

（1）安装。

```
root@ghostcloud:~# apt-get install docker-engine
```

（2）启动 Docker Daemon。

```
root@ghostcloud:~# service docker start
```

（3）测试安装是否成功。

```
root@ghostcloud:~# docker version
Client:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:54:52 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:54:52 2016
 OS/Arch:      linux/amd64
```

（4）运行第一个容器。

```
root@ghostcloud:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

以上步骤首先启动了一个容器，然后打印出了一些语句，最后容器退出。

2.2　在 CentOS 下安装

CentOS 是 Red Hat Enterprise Linux 的开源版本，也是使用非常广泛的版本。Docker 可以运行在 CentOS 7.X 上，本节讲述如何在 CentOS 上安装 Docker。

2.2.1　前置条件

同 Ubuntu 一样，依然需要内核版本高于 3.10 的 64 位系统，通过 uname-r 可以检查。

```
root@ghostcloud:~# uname -r
3.10.0-229.el7.x86_64
```

2.2.2　更新 yum

yum 是 CentOS 的包管理工具，类似于 apt。以 root 用户登录系统，然后更新 yum 源。

```
root@ghostcloud:~# yum update
```

2.2.3　添加仓库

在/etc/yum.repos.d/ghostcloud.repo 中加入：

```
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
```

2.2.4　正式安装

（1）安装。

```
root@ghostcloud:~# yum install docker-engine=
```

（2）启动 Docker Daemon。

```
root@ghostcloud:~# service docker start
```

（3）测试安装是否成功。

```
root@ghostcloud:~# docker version
Client:
 Version:      1.10.3
 API version:  1.22
```

```
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:54:52 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:54:52 2016
 OS/Arch:      linux/amd64
```

（4）运行第一个容器。

```
root@ghostcloud:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

以上步骤启动了一个容器，然后打印出了一些语句，最后容器退出。

2.3　通过 Ghostcloud 进行安装

前几节讲述的安装步骤其实是非常烦琐的，需要用户自己去更新源，而且由于国外网络的原因，有时候可能会失败。因此，国内出现了一些 PaaS 服务商将 Docker 源在国内做了一个镜像，同时提供单个命令进行安装。目前，www.ghostcloud.cn 是安装速度最快的平台，本书选用它来进行安装。

2.3.1　注册 Ghostcloud 账号

登录 www.ghostcloud.cn，注册用户。每个新用户账户上都有 50 元用于免费体验。点击进入“控制台”，如图 2.1 所示。

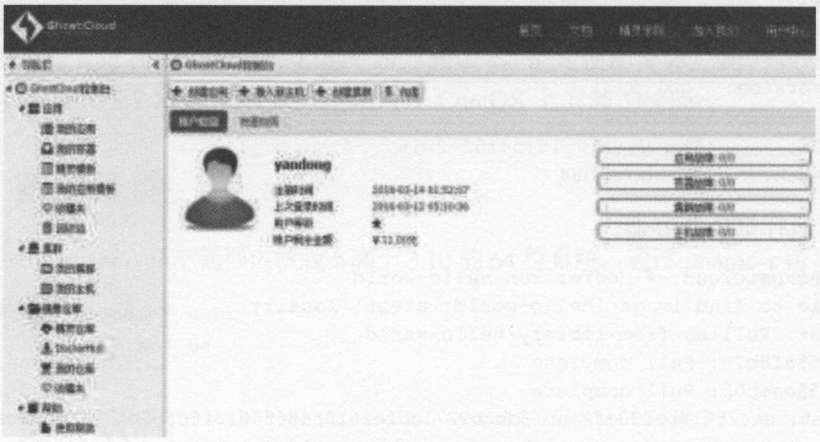


图 2.1　Ghostcloud 控制台

2.3.2　接入新主机

选择接入新主机，如图 2.2 所示。

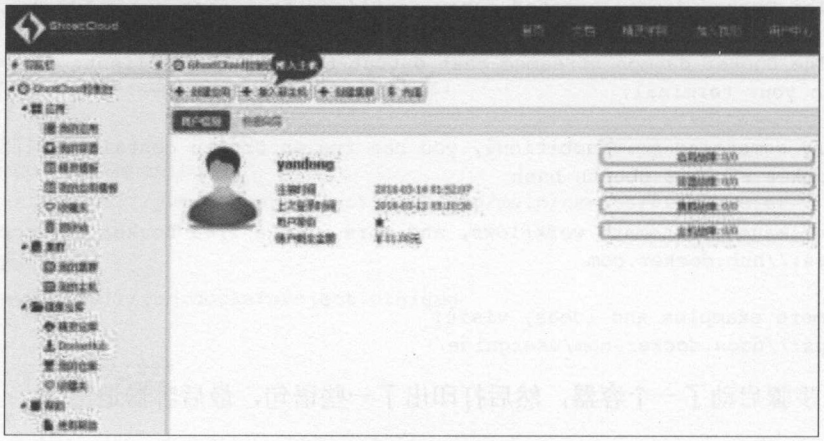


图 2.2　接入新主机

2.3.3　获取安装脚本

选择默认选项后，进入获取安装脚本页面，如图 2.3 所示。

复制添加主机的命令，并在主机上执行。

```
root@ghostcloud:~# curl -Ls http://release.ghostcloud.cn/install/ install_
agent.sh | sudo -H sh -s 01de123f-e9b2-11e5-b5f4-0242c0a80002d
```

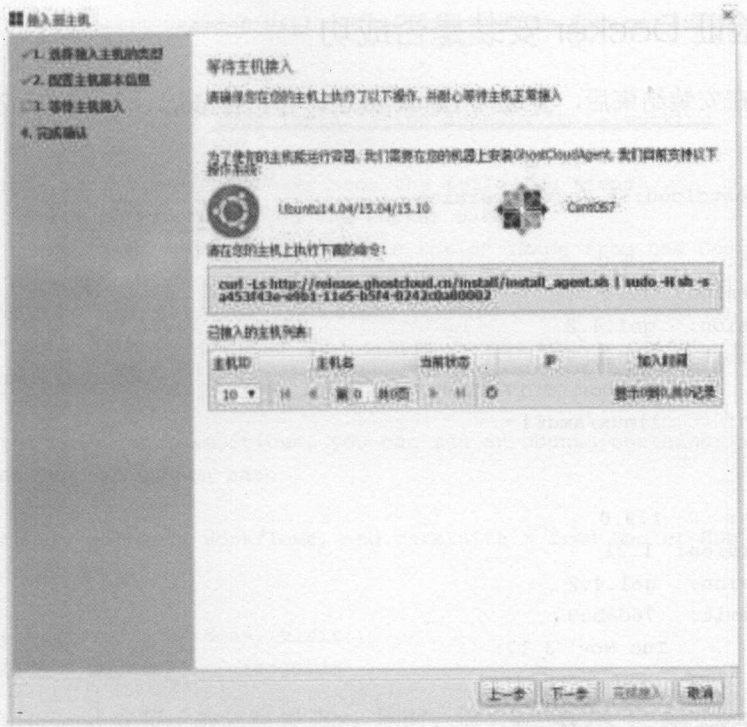


图 2.3　获取安装脚本

安装成功后，Ghostcloud 界面会有提示，同时接入到统一管理平台，如图 2.4 所示。

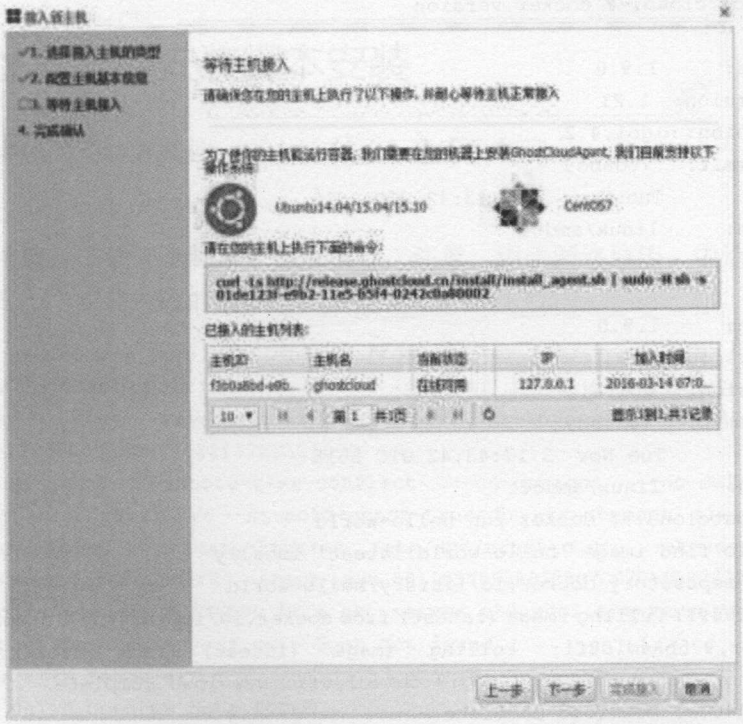


图 2.4　接入到平台

2.3.4　验证 Docker 安装是否成功

通常，在安装结束后，会进入 Linux 终端，并执行 docker version 或 docker info 命令。

```
root@ghostcloud:~# docker version
Client:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:43:42 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:
```

2.3.5　运行第一个容器

同样，运行第一个容器。

```
root@ghostcloud:~# docker version
Client:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:43:42 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:43:42 UTC 2015
 OS/Arch:      linux/amd64
root@ghostcloud:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
Pulling repository docker.io/library/hello-world
975b84d108f1: Pulling image (latest) from docker.io/library/hello-world, mirror:
http://mirror.975b84d108f1:  Pulling  image  (latest)  from  docker.io/library/
hello-world, endpoint: https://regi975b84d108f1: Download complete
3f12c794407e: Download complete
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world: this image was pulled from a legacy registry.
```

```
Important: This registry version will not be supported in future versions of docker.

    Hello from Docker.
    This message shows that your installation appears to be working correctly.

    To generate this message, Docker took the following steps:
     1. The Docker client contacted the Docker daemon.
     2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
     3. The Docker daemon created a new container from that image which runs the
        executable that produces the output you are currently reading.
     4. The Docker daemon streamed that output to the Docker client, which sent it
        to your terminal.

    To try something more ambitious, you can run an Ubuntu container with:
     $ docker run -it ubuntu bash

    Share images, automate workflows, and more with a free Docker Hub account:
     https://hub.docker.com

    For more examples and ideas, visit:
     https://docs.docker.com/userguide/
```

通过 Ghostcloud 安装一般只需要 20s。同时，该公司提供了仓库的 mirror，可以大大提高 pull 镜像的速度。如果不进行集群管理，则可以只使用命令行，如果需要进行集群管理，则可以研究一下他们的管理平台，在这里不做赘述。

2.4　通过官方的安装脚本安装

除以上讲述的安装方式外，官方还有一个通用安装脚本：

```
# curl -fsSL https://get.docker.com/ | sh
```

这行命令就是下载一个脚本，然后执行。注意，由于网络原因，中途很可能会失败。以下就是一个安装失败的例子。

```
root@ghostcloud:~# curl -fsSL https://get.docker.com/ | sh
apparmor is enabled in the kernel and apparmor utils were already installed
+ sh -c apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring
--homedir /tmp/tmp.cr8hxY45Ve --no-auto-check-trustdb --trust-model always --keyring
/etc/apt/trusted.gpg --primary-keyring /etc/apt/trusted.gpg --keyserver hkp://
p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
gpg: requesting key 2C52609D from hkp server p80.pool.sks-keyservers.net
?: p80.pool.sks-keyservers.net: Host not found
gpgkeys: HTTP fetch error 7: couldn't connect: Success
gpg: no valid OpenPGP data found.
gpg: Total number processed: 0
```

2.5　在非 Linux 系统下安装 Docker

由于 Docker 都需要 Linux 内核的支持，如果要在非 Linux 环境下安装 Docker，基本上都需要通过虚拟机的形式来进行。其原理就是，在非 Linux 系统上安装一个客户端，然后连接到虚拟机里操作，所以这种方式其实并不是原生态的安装。Docker Machine 就是专门为解决这种问题而生的。目前，Mac OS X 和 Windows 操作系统下，都可以安装 Docker Machine，而且 Docker 将其封装得非常易用，基本上使用户感知不到与原生态的差别，不过这只能运用在测试和开发环境中。

尽管 Docker Machine 提供了跨平台的非 Linux 系统支持，但是其他操作系统也是可以支持 Docker 的。目前，Windows Server 2016 已经提供了对 Docker 的全套支持，有兴趣的读者可以在网上搜索相关的视频教程。

如图 2.5 所示是原生态的 Docker，使用 Docker Machine 的结构示意图如图 2.6 所示。

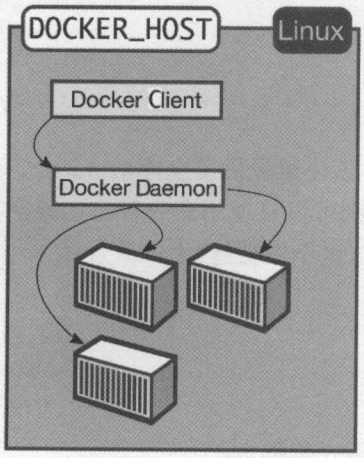


图 2.5　Linux 原生构架

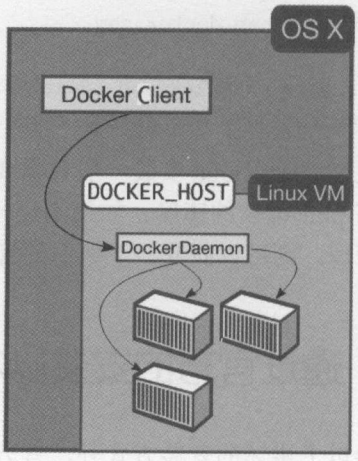


图 2.6　Docker Machine 架构图

如果需要使用 Docker Machine，可以在 https://www.docker.com/toolbox 进行下载安装。

2.6　习题

本章介绍了在 Ubuntu 和 CentOS 上安装 Docker 的步骤，并启动容器。接下来，通过习题和实验检验本章的学习成果。

（1）在内网中安装一个能够运行 Docker 的操作系统，然后安装 Docker。

（2）在公有云中，运行一个能够运行 Docker 的操作系统镜像，然后安装 Docker。

（3）在 Windows 环境下，通过 Docker Machine，安装 Docker。

（4）运行一个 hello-world 容器。

第3章　使用Docker

通过前面的章节，已经顺利地安装了Docker。接下来，就真正开始使用Docker。曾经有一个笑话，某天一个程序员决定练书法，他拿起毛笔，精神抖擞地写了两个大字——Hello World。如何使用Docker也是从Hello World开始。

本章主要涉及的知识点有：

- 通过hello-world镜像学习最基本的容器运行流程
- 学习Docker容器和镜像
- Docker的入门操作

3.1　运行hello-world

hello-world镜像是Docker官方推荐的第一个学习镜像，它的功能很简单，运行后将输出一段日志在终端上。本节将以该镜像为基础，详细介绍如何运行第一个Docker容器。

Docker是基于Linux内核的Namespace、CGroups和UnionFS三项基本功能实现的，因此，运行Docker容器需要进入Linux的终端，并执行下面一句命令，执行后会出现后面的信息。

```
root@ghostcloud:~# docker run hello-world

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
```

```
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

这条命令包括了三部分：

- docke——Docker 的客户端程序；
- run——子命令，用于运行容器；
- hello-world——镜像的名称。

这条命令首先会在本地查找是否有 hello-world 这个镜像，如果没有，则会自动从 Dockerhub（Docker 主仓库）去拉取，之后会启动一个容器，并把镜像装载进容器中运行。所有 Docker 的命令格式都是下面的格式：

```
docker [OPTIONS] COMMAND [arg...]
```

其中，OPTIONS 是运行的参数，COMMAND 是真正的子命令，arg 是该条子命令对应的参数集合。

下面，通过 docker images 来看一下系统中都有哪些镜像。

```
root@ghostcloud:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos              latest              eb98cdc284d4        9 days ago          196.6 MB
ubuntu              latest              56063ad57855        10 days ago         188 MB
hello-world         latest              975b84d108f1        5 months ago        960 B
```

在该系统中存在三个镜像，分别是 centos、ubuntu、hello-world。每列的具体意思是什么呢？

- REPOSITORY——这是镜像的名称，但是官方取的名字有点奇怪。
- TAG——这是镜像的版本，类似于 git 的 tag，默认情况是，如果 run 时不带版本就会拉取最新的版本 latest。
- IMAGE ID——这是镜像的截短后的 ID。
- CREATED——镜像是在什么时候创建的，这个时间是指的仓库里面的时间。
- VIRTUAL SIZE——这是这个镜像所占空间的大小。

3.2　容器和镜像

容器和镜像是 Docker 最核心的部分，使用 Docker 时实际就是在容器里面运行一个镜像，本节将分别对两者做介绍。

3.2.1　什么是容器

很多用户在接触 Docker 之初都会认为容器就是一种轻量级的虚拟机，但实际上，容器和虚拟机有非常大的区别。从根本形态上来看，容器其实就是运行在操作系统上的一个进程，只不过加入了对资源的隔离和限制。在学习操作系统时，都会学习什么是进程，简单地讲，进程就是一个运行的程序。传统上，在运行一个进程时，如果里面出现死循环，CPU 就会一下被占用完；如果出现内存泄露或者大内存分配，就可能会把系统的内存用完，这是因为默认进程间共用了 CPU 和内存。但是这种不进行任何隔离的处理方式，就会遇到相互间干扰的问题，在企业级产品环境中，这一点是非常致命的，任何一个小程序都可能导致整个系统的不可用。因此，早在 2006 年就出现了进程间的资源隔离技术，后来 Linux 也同样有了类似的实现。最初，Linux 的容器技术是基于 LXC 的，Docker 在易用性和稳定性方面做了很大改善，其三大核心功能就是 CGroups、Namespace 和 UnionFS。CGroups 技术用来限定一个进程的资源使用；在一个操作系统之上，用户 ID、机器名等资源是全局的，运行的进程间都是访问同一份资源，为了达到隔离的目的，Linux 又出现了 Namespace 技术用来划分不同的命名空间；而 UnionFS 则是用来处理分层镜像的功能。

说明　LXC 是 Docker 早期使用的技术，后来 Docker 自行开发了 libcontainer，二者都是对 Linux 内核功能的封装。

在安装了 Docker 的机器上，可以通过 run 命令启动容器，并通过 ps 命令查看已存在的容器。

3.2.2　什么是镜像

容器是一个动态的概念，而镜像是一个相对静止的概念。简单来说，镜像就是容器中的文件系统。早在 1980 年就出现了文件系统管理技术，那就是 chroot 系统调用。通过该技术可以改变进程运行的工作目录，并将其限定在这个目录中，不过它只能做简单的隔离，而且存在安全隐患。因此，Docker 就使用了，注意不是设计了一个 Layered FS，它把文件系统分成多个层，使多个容器间可以使用公共的部分。而镜像就是由 Layered FS 组成的，并且它是只读的。当容器运行时，会在镜像之上再加上一层可读

可写层。

> **说明**　镜像除了包含文件系统，还集成了一部分容器运行的参数，可以将镜像看作容器的模板。

3.2.3　容器和镜像的关系

容器和镜像是密切相关的，二者缺一不可，也是可以相互进行转换的，如图 3.1 所示。

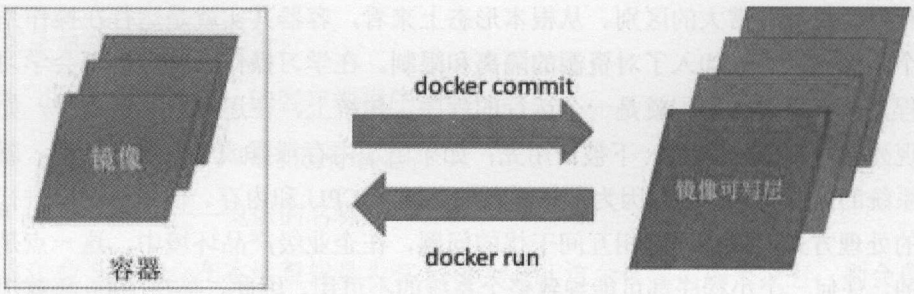


图 3.1　容器和镜像的关系

Docker 的镜像最初是从官方的仓库拉取获得的，之后通过 docker run 即可启动一个容器运行。当需要将容器转化为镜像时，可以通过 docker commit 进行转化。

3.3　Docker 入门操作

本节将通过一个实验更近一步地了解如何使用 Docker，期间涉及一些新的 Docker 命令：

- docker info；
- docker pull；
- docker ps；
- docker start/stop。

3.3.1　查看 Docker 基本信息

docker info 是经常用来查看运行状态及版本信息的命令，是整个 Docker Daemon 守护进程运行状况的缩影，包括容器个数、镜像个数、Daemon 版本、使用的存储驱动等信息。如果 Docker 安装成功，执行命令后则会有以下信息。

```
root@ghostcloud:~# docker info
```

```
Containers: 6
Images: 15
Server Version: 1.9.0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 31
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-25-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 2
Total Memory: 3.844 GiB
Name: ghostcloud
ID: I37F:V3WA:R5OW:J7CU:3YPU:MLYI:AKVN:6EAG:CKLE:H6NS:MQCQ:7WCW
WARNING: No swap limit support
root@ghostcloud:~#
```

如果安装失败，执行命令后会出现以下信息：

```
root@ghostcloud:~# docker info
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

3.3.2　下载第一个基础镜像

Docker 的镜像是分层的，因此使用时都需要从某一个基础镜像开始。官方推荐的基础镜像是 Ubuntu 14.04，我们可以通过 docker pull 从 Docker Hub 进行拉取。如果已经存在，命令会直接返回，否则会自行下载。但是，国内访问国外的镜像比较慢，推荐用户使用国内的镜像仓库，如 Ghostcloud、DaoCloud 等。下面是拉取镜像的实例。

```
root@ghostcloud:~# docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu

073de23ee32b: Downloading 15.28 MB/65.69 MB
073de23ee32b: Pull complete
9d89fd8f8a3e: Pull complete
0b427fcc4cbb: Pull complete
8ed581e3fa7a: Pull complete
Digest: sha256:4e85ebe01d056b43955250bbac22bdb8734271122e3c78d21e55ee235fc6802d
Status: Downloaded newer image for ubuntu:latest
```

注意　有时候由于网速和防火墙原因，拉取可能会失败，可以重复尝试。

3.3.3　运行一个含 shell 终端的容器

在 hello-world 例子中，容器运行完成后就自动退出了，那么可不可以运行一个可交互的容器呢，就像是一个虚拟机一样。答案是可以的。

```
root@ghostcloud:~# docker run -i -t ubuntu /bin/bash
root@ee87f6127814:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@ee87f6127814:/# pwd
/
root@ee87f6127814:/# touch 1.txt
root@ee87f6127814:/# ls
1.txt  boot  etc   lib    media  opt   root  sbin  sys  usr
bin    dev   home  lib64  mnt    proc  run   srv   tmp  var
```

看见了吗，就是这么简单，一键就拥有了一个 Ubuntu 系统，这就是容器的魅力。从现在开始，将正式进入容器的世界。首先来剖析运行的参数：

- -i——表示启动一个可交互的容器；
- -t——表示使用 pseudo-TTY，关联到容器的 stdin 和 stdout；
- ubuntu——表示要运行的镜像；
- /bin/bash——表示启动容器时要运行的命令。

3.3.4　查看容器运行

由于 3.3.3 节是通过交互模式运行的容器，当前终端就会被占据，为了查看容器运行，必须开启一个新的终端查看容器的运行状态，如图 3.2 所示。

图 3.2　查看运行中的容器

注意　如果在交互式容器中执行 exit 命令，那么整个容器就会停止运行，如果需要退出交互模式且不影响容器运行，可以通过【Ctrl+PQ】进行。

3.3.5　运行长时间容器

通过 hello-world 和 Ubuntu 的运行，应该可以发现，容器会随着执行的命令结束而结束，这也是为什么很多用户会发现运行了容器，但是 ps 却没有的原因。接下来，可以运行一个长时间的容器，如图 3.3 所示。

```
root@ghostcloud:~# docker run -d ubuntu /bin/sh -c "while true; do echo Hello world; sleep 1; d
one"
be9c863ca0e39548e44a062aaead3b7894f135fbc05e3109d055efeabd3bccba
root@ghostcloud:~#
root@ghostcloud:~# docker logs be9c
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
root@ghostcloud:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
        PORTS               NAMES
be9c863ca0e3        ubuntu              "/bin/sh -c 'while tr"   24 seconds ago      Up 23 seco
nds                 trusting_franklin
ee87f6127814        ubuntu              "/bin/bash"              9 minutes ago       Up 9 minut
es                  pensive_booth
root@ghostcloud:~#
```

图 3.3　长时间运行容器

通过命令使容器不停地循环打印 hello world：

- -d——让容器在后台运行，不关联到当前终端的 shell 上；
- docker logs <container_id>——查看容器的日志，其实就是查看容器的标准输出日志。

为了避免产生过多的日志，可以通过 docker kill <container_id>将其杀掉。

3.3.6　查看所有容器

可以使用 ps 命令查看容器，但是当容器退出后，该命令就列举不出容器，如果要查看所有容器，则需要借助-a 参数，如图 3.4 所示。

```
root@ghostcloud:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
    PORTS               NAMES
ee87f6127814        ubuntu              "/bin/bash"         13 minutes ago      Up 13 minutes
                    pensive_booth
root@ghostcloud:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
                    PORTS               NAMES
be9c863ca0e3        ubuntu              "/bin/sh -c 'while tr"   4 minutes ago       Exited (13
7) About a minute ago                   trusting_franklin
ee87f6127814        ubuntu              "/bin/bash"              13 minutes ago      Up 13 minu
tes                                     pensive_booth
db2bf300f148        hello-world         "/hello"                 About an hour ago   Exited (0)
 About an hour ago                      sick_boyd
7316d8dc8377        hello-world         "/hello"                 2 hours ago         Exited (0)
 2 hours ago                            gloomy_kare
274edbe026b3        centos              "/bin/bash"              3 hours ago         Exited (0)
 2 hours ago                            loving_hawking
e314f0a8ec31        56063ad57855        "/bin/bash"              3 hours ago         Exited (0)
 3 hours ago                            nostalgic_darwin
e5ca6bd8b515        hello-world         "/hello"                 3 hours ago         Exited (0)
 3 hours ago                            angry_lovelace
5ed09e80015b        56063ad57855        "/bin/bash"              2 days ago          Exited (10
0) 6 hours ago                          clever_meninsky
root@ghostcloud:~#
```

图 3.4　查看所有容器

注意　通过该命令，可以发现容器退出后，它本身还存在于主机上，因此一定要对不需要的容器进行清理，否则将占据很大的磁盘空间。

3.4　习题

本章介绍了容器是什么，镜像是什么，以及它们之间的关系。同时，介绍了 Docker 的基础命令。接下来，通过习题和实验检验本章的学习成果。

（1）讲述 Docker 容器和虚拟机的区别。

（2）如何将容器转化成镜像？

（3）列举出本机的所有容器及所有镜像。

第 2 篇

Docker 的基本使用

第 4 章　Docker 深入解析

常用的 Docker 命令主要包括：

- docker pull——拉取镜像；
- docker run——运行容器；
- docker images——列出所有镜像；
- docker ps——查看容器；
- docker stop/start/restart/kill——容器的启停；
- docker logs——查看容器的日志。

接下来，本章要对 Docker 命令的运行机制进行讲解。

4.1　Docker 的架构

Docker 使用的是 C/S 架构（如图 4.1 所示）。Docker Client 同 Docker Daemon 进行交互，其中主要的工作是通过 Daemon 来完成，包括拉取镜像、编译镜像、运行容器、发布容器等。Docker Client 和 Docker Daemon 可以运行在同一个系统上，也可以通过远程方式进行访问。Docker Client 和 Docker Daemon 之间是在 Socket 上通过 RESTful API 进行交互的。

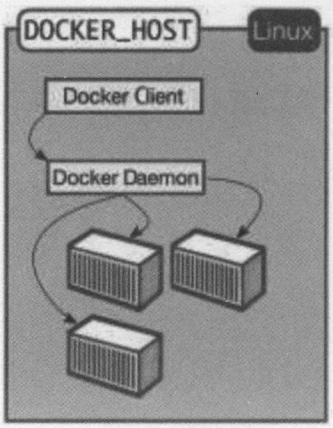


图 4.1　Docker 的架构

1. Docker Daemon

如图 4.1 所示，Docker Daemon 运行在一个主机上，用户并不是直接同 Docker Daemon 进行交互，而是通过 Docker Client 进行访问。

2. Docker Client

Docker Client 是主要的用户访问 Docker 的渠道。用户通过它对 Docker Daemon 进行访问控制。

3. Docker Image

Docker Image 是一个只读的模板。例如，一个 Image 可以包含一个 Ubuntu 操作系统，整个系统可以包括 Apache 和 Web 应用。Image 是用来创建容器的。Docker 提供了一种简单的方式来创建 Image 和更新已有的 Image，用户可以从网上下载 Image，也可以自己编译 Image。

4. Docker Registry

Docker Registry 是存放 Image 的仓库。我们可以使用公有的和私有的 Registry 来进行下载和上载。公共的 Docker Registry 位于 Docker Hub，但是国内访问比较慢。Docker Hub 包含了大量已有的 Image，供用户使用。你可以基于之前的 Image 来创建自己的 Image。

5. Docker Containers

Docker 容器就像一个文件夹。一个容器包含了应用程序运行所需的所有环境。每个容器都源于某一个 Docker Image。Docker 容器可以运行、开始、停止、移动并删除。每个容器都是完全隔离的、安全的应用。

4.2 Docker 如何工作

通过之前的介绍，可以知道：

- 可以自己编译 Docker Image 来打包应用；
- 可以从 Docker Image 创建容器，并在其中运行应用程序；
- 可以通过 Docker Hub 或私有的 Registry 分享 Docker Image。

接下来，将讲述如何将之前的这些元素整合起来运行。

4.2.1 Docker Image 工作方式

Docker Image 是只读模板，并随容器一起启动。每个镜像包含多个层。Docker Image 使用的是 Union File System 来将这些层组合成一个镜像。Union File System 可以将文件和目录（通常称作 Branch）进行透明的层叠组装，然后形成一个单独的文件系统。Docker 为什么轻量，就是因为使用了这些层状的文件系统。当用户修改一个 Docker Image 时（如更新应用程序）一个新的层就会被建立。因此，这是一种增量式的修改，而不是新建一个全新的镜像，这也是区别于传统虚拟机的特点。当发布一个新的镜像时，只需发布差异的部分，因此速度就非常快。

每个镜像都来自于一个最基础的镜像，如 Ubuntu 是一个基础镜像，Fedora 也是一个基础镜像。用户也可以使用自己的镜像作为基础镜像，如可以创建包含了一个 Apache 服务器的镜像，作为所有 Web 应用的基础镜像。

注意　Docker 通常从 Docker Hub 获取基础镜像。

Docker_Image 都是从这些基础镜像衍生而来的，编译镜像是由一系列指令组成的，每个指令在我们的镜像上创建一个新的增量。指令包括：

（1）执行一条命令；

（2）添加文件或文件夹；

（3）创建环境变量；

（4）容器启动时，运行什么程序。

4.2.2　Docker Registry 工作方式

Docker Registry 是镜像的仓库，你编译完成一个镜像时，可以推送到公共的 Registry，如 Docker Hub，也可以推送到自己的私有 Registry。使用 Docker Client，可以搜索已经发布的镜像，并从中拉取镜像到本地，并在容器中运行。Docker Hub 提供了公有和私有的 Registry。所有人都可以搜索和下载公共镜像。私有仓库只有私有用户能够查询和下载。

4.2.3　容器工作方式

一个容器由操作系统、用户文件和元数据构成。由此可知，每个容器都根据镜像来生成。这个镜像告诉 Docker 容器包含什么内容，运行什么程序，以及其他配置信息。Docker Image 是只读的，当一个容器运行一个镜像时，容器会在 Union FS 的顶层增加文件层。

假如运行下面一条指令，执行后出现下面的信息。

```
root@gctest:~# docker run -i -t ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
Digest: sha256:f91f9bab1fe6d0db0bfecc751d127a29d36e85483b1c68e69a246cf1df9b4251
Status: Downloaded newer image for ubuntu:latest
root@ea7ffc10c7e7:/#
```

Docker Client 通过 run 命令告诉 Daemon 启动一个新的容器。这个指令至少需要包括：

（1）需要运行什么 Image，这里使用的是 Ubuntu 基础镜像。

（2）需要在容器启动时运行什么命令，这里使用的是/bin/bash——是否需要进入应

用程序，这里指定的是-i -t，就是进入容器交互模式。

那么具体到内部的流程是怎么样的呢？

（1）拉取 Ubuntu 镜像：Docker 检查本地是否有 Ubuntu 镜像，如果不存在就自动从 Docker Hub 拉取。如果存在就进入下一步。

（2）创建一个容器：一旦本地存在 Ubuntu 镜像，Docker 将通过它来创建容器。

（3）分配文件系统并挂载一个 RW 层：容器是创建在文件系统中的，并且在其之上增加了一层读写层。由此可以看出容器并不会改变原始的镜像。

（4）分配网络/桥接模式：创建一个桥接网络接口，使容器可以和本地主机进行通信。

（5）设置一个 IP 地址：根据本地网络情况，选取一个可用的 IP 挂载到容器之上。

（6）启动一个进程：这里就是/bin/bash。

（7）抓取应用程序的输出：将程序的 stdin、stdout 和 stderr 进行捕捉，这样就可以看到程序的运行情况。至此，就拥有了一个运行的容器。通过容器，可以运行程序，并且进行交互。当程序执行完毕，可以停止和删除程序。

4.2.4　底层的技术

Docker 是用 Go 编写的，同时使用了多种内核的功能实现我们现在所看到的功能。

1. Namespaces

Docker 使用了 Namespace 技术来隔离工作区，也就是通常所说的容器。当容器运行时，Docker 创建了一系列的 Namespace。通过 Namespaces，容器运行在它自己的独立命名空间之中，而外层没有访问权限。目前，Docker 使用了以下 Namespace:

- PID Namespace——用于进程的隔离(PID: Process ID)；
- NET Namespace——用于管理网络接口 x(NET: Networking)；
- IPC Namespace——用于管理进程间通信(IPC: Inter Process Communication)；
- MNT Namespace——用于管理 Mount 点(MNT: Mount)；
- UTS Namespace——用于隔离内核和版本信息(UTS: UNIX Timesharing System)。

2. Control Groups

Docker 也使用了 CGroups 这项内核技术，通过 CGroups 可以限制应用程序使用的资源，这项技术可以使用户主机更好地运行多个容器而相互间不受影响。CGroups 可以限定容器使用的硬件资源，如内存数量、CPU 数量等。

3. Union File System

Union FS 用来对文件系统进行分层，通过分层可以使镜像更加轻量级和快速。Docker 可以使用多种不同的 Union FS，如 AUFS、Btrfs、VFS、DevicemapperFS 等。

4.3 Docker Client 和 Daemon

Docker-engine 在主机上是 C/S 架构，其中，C 指的是 Docker Client，S 指的是 Docker Daemon，但是这二者使用的是同一个程序。

```
root@ghostcloud:~# which docker
/usr/bin/docker
```

可以通过 docker --help 查看 Docker 的所有命令及选项：

```
root@ghostcloud:~# docker --help
Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:

  --config=~/.docker                Location of client config files
  -D, --debug=false                 Enable debug mode
  --disable-legacy-registry=false   Do not contact legacy registries
  -H, --host=[]                     Daemon socket(s) to connect to
  -h, --help=false                  Print usage
  -l, --log-level=info              Set the logging level
  --tls=false                       Use TLS; implied by --tlsverify
  --tlscacert=~/.docker/ca.pem      Trust certs signed only by this CA
  --tlscert=~/.docker/cert.pem      Path to TLS certificate file
  --tlskey=~/.docker/key.pem        Path to TLS key file
  --tlsverify=false                 Use TLS and verify the remote
  -v, --version=false               Print version information and quit

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders between a container and the local filesystem
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Export a container's filesystem as a tar archive
    ...

Run 'docker COMMAND --help' for more information on a command.
```

首先看一下头几行。

```
Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ --help | -v | --version ]
```

（1）第一行是 Docker Client 的用法，还可以通过 docker COMMAND --help 来查看每个命令的详细用法。

（2）第二行是 docker daemon 命令的用法，可以通过 docker daemon --help 来查看。

```
root@ghostcloud:~# docker daemon --help

Usage:  docker daemon [OPTIONS]

Enable daemon mode

  --api-cors-header=                  Set CORS headers in the remote API
  -b, --bridge=                      Attach containers to a network bridge
  --bip=                            Specify network bridge IP
  --cluster-advertise=                Address or interface name to advertise
  --cluster-store=                   Set the cluster store
  --cluster-store-opt=map[]           Set cluster store options
  -D, --debug=false                  Enable debug mode
  …
```

这里面包含了很多 Docker 可以配置的选项，可以停止服务，直接使用程序方式带参数运行，也可以修改服务的配置文件/etc/default/docker 来加入默认配置。如果要查看 client 和 daemon 的信息，可以使用 docker version 来查看。

```
root@ghostcloud:~# docker version
Client:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:43:42 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:43:42 UTC 2015
 OS/Arch:      linux/amd64
```

4.4　通过容器运行 Web 应用

容器可以运行任何的 Linux 程序，接下来将开启互联网之旅，实验一下容器运行 Web 应用。本节将使用一个国内的仓库，而不是 Docker 的主仓库。

4.4.1　使用国内仓库

由于国外的 Docker Hub 访问速度非常缓慢，经常会拉取不下来镜像，所以在本节将使用由 Ghostcloud 提供的 Docker 仓库。如果通过 Ghostcloud 安装 Docker，就可以默认使用 hub.ghostclou.cn 的仓库，否则需要修改一下本机的配置，将

```
--insecure-registry=hub.ghostcloud.cn
```

添加到在/etc/default/docker 的 DOCKER_OPTS 中，并重启 Docker Daemon。

4.4.2　拉取 apache-php 镜像

拉取镜像是通过 docker pull 命令来进行的，镜像名称可以带域名也可以不带，不带域名则默认从 Docker Hub 拉取，否则从指定的域名拉取。

```
root@ghostcloud:~# docker pull hub.ghostcloud.cn/apache-php
Using default tag: latest
latest: Pulling from apache-php
b3efe11ed0e2: Pull complete
dacf881c67ea: Pull complete
1d5466c852f4: Pull complete
a053e7ea2233: Pull complete
95893f055a98: Pull complete
5ab76f7b9b5a: Pull complete
5f426bf84ca4: Pull complete
7f9885e7d7c3: Pull complete
ad7ca9b768a8: Pull complete
131a92c5d64b: Pull complete
Digest: sha256:3ffadad4c12bdd37ada051ce5316dcdfd31979d93df25252147f277f5ff876c8
Status: Downloaded newer image for hub.ghostcloud.cn/apache-php:latest
```

在本例中，拉取的是 hub.ghostcloud.cn 的 apache-php 镜像。

4.4.3　运行镜像

运行容器是通过 docker run 命令进行的，同时可以指定一些参数。

```
root@ghostcloud:~# docker run -d -p 80:80 hub.ghostcloud.cn/apache-php

0a899413049f4f68963c5dba5c05be231a90269547eeca0bb73ad1c61b1ee512
```

说明　-p 是指定主机和容器的端口映射，-d 是在后台运行，不关联到当前终端。

4.4.4　网页访问

访问容器的网页时，默认需要通过访问主机的端口，然后映射到容器端口，上例中是将主机的 80 端口映射到容器的 80 端口，因此访问主机的 80 端口即可访问到容器中运行的网页。在这里，服务器 IP 地址为 192.168.10.10，因此访问地址是：http://192.168.

10.10:80，由于 http 默认就是 80 端口，所以可以省略掉端口，通过 http://192.168.10.10 进行访问，如图 4.2 所示。

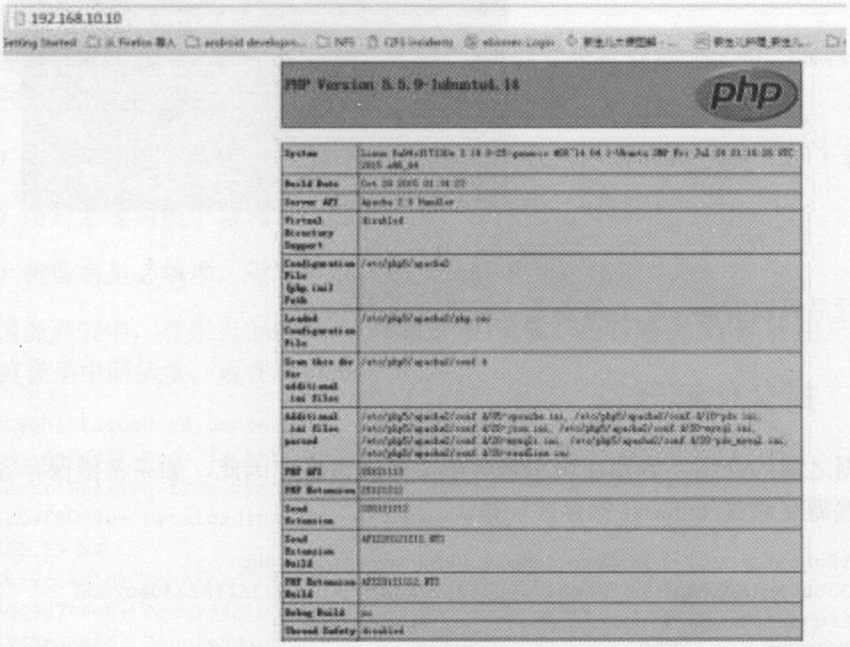


图 4.2　访问容器中的网页

4.4.5　修改页面内容

下面将进入容器修改默认的页面。

（1）通过 exec 进入容器。

```
root@ghostcloud:~# docker exec -it 0a8 /bin/bash
root@0a899413049f:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
```

说明　docker exec 可以在指定的容器中运行命令，这里，容器 ID 是 0a8，运行的命令是/bin/bash。如果不加-it 参数，执行完成后就会退出，所以需要通过-it 进入容器，这也是标准的无干扰进入容器的方法。

（2）进入网页目录/var/www/html，并修改 index.php。

```
Hello World
<?php
phpinfo();
?>
```

在第一行加入 Hello World。

（3）通过网页再次访问，如图 4.3 所示。

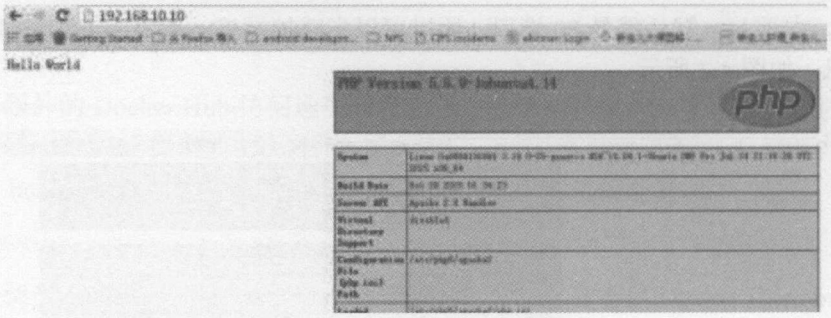


图 4.3　修改后的网页

运行后可以看到，左上角多了一句“Hello World”。

4.4.6　持久化容器

根据之前的介绍，容器在退出后并不会更改镜像。因此，如果希望保存容器中的数据，就需要通过 commit 保存成镜像。

```
root@ghostcloud:~# docker commit 0a89 my-apache-php
99a35bc09c482164bfd05f20e9477303531f4c10f7310f0da361f92330ac2b03
root@ghostcloud:~# docker images my-apache-php
REPOSITORY       TAG             IMAGE ID        CREATED          VIRTUAL SIZE
my-apache-php    latest          99a35bc09c48    28 seconds ago   250.3 MB
```

之后，my-apache-php 中就都默认包含了之前的改动，可以通过 docker run 进行尝试。

4.5　镜像制作

从本节开始将系统性地介绍镜像的一些基础概念。镜像是容器的基础，当运行 docker run 时，通常会指定一个镜像，这个镜像一方面承载了容器的文件系统，同时也包含了容器启动时需要执行的入口文件等信息。首先来熟悉一下镜像的基础命令。

4.5.1　查看本机镜像

docker images 是查看本机镜像列表的方法，如图 4.4 所示。

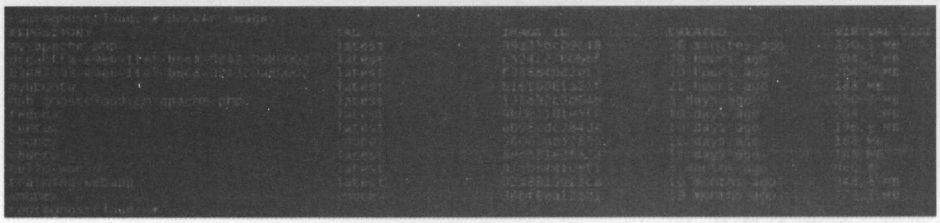


图 4.4　查看本机镜像

4.5.2　获取镜像的三种方式

1．拉取镜像

这种方式会接收一个镜像名作为参数，但是镜像名是有特定含义的，其格式为：[域名/][用户名/]镜像名[:版本号]。

（1）当用户指定了域名时，会从该域名进行下载，否则从 Docker Hub 下载。

（2）用户名是隶属于该域名下的子目录，在使用私有仓库时有用。

（3）镜像名是必填项，版本号默认是 latest，也可以指定。

在镜像列表中，有来自 hub.ghostcloud.cn 的镜像，也有属于 training 用户的镜像，<none>镜像是中间镜像，现在可以不用管。

```
root@ghostcloud:~# docker pull hub.ghostcloud.cn/mysql
Using default tag: latest
latest: Pulling from mysql
044ffdf80f70: Downloading [==============>                                    ] 14.46
MB/51.37 MB
9d6be9c04d1c: Download complete
555dd9d744e8: Download complete
5d9945c35afd: Download complete
b801734f5057: Download complete
4aa7ca2c1a59: Download complete
01da3cd98fb7: Download complete
c6283fb42105: Download complete
838b735ebc0b: Download complete
e9c6b2cc4b90: Downloading [===========>                                       ] 14.3
MB/63.96 MB
0a983fabb1e9: Download complete
ac1f583509c9: Download complete
f2784f200768: Download complete
4b93b18502fe: Download complete
7ca99f1ad085: Download complete
7ce9d75f98f0: Download complete
```

2．把容器转换成镜像

通过 commit 把一个容器转换成镜像。

3．制作镜像

通过 Dockerfile 生成镜像，Dockerfile 类似 Makefile，也加入了一些制作镜像的指令。本书将在后续章节介绍 Dockerfile 的用。Dockerfile 在持续集成中有着非常广泛的应用。

4.5.3　查找 DockerHub 镜像

查找默认是从 Docker Hub 进行搜索。

```
root@ghostcloud:~# docker search
```

```
docker: "search" requires 1 argument.
See 'docker search --help'.

Usage:  docker search [OPTIONS] TERM

Search the Docker Hub for images
```

搜索 Docker Hub 中的所有 MySQL 镜像，如图 4.5 所示。

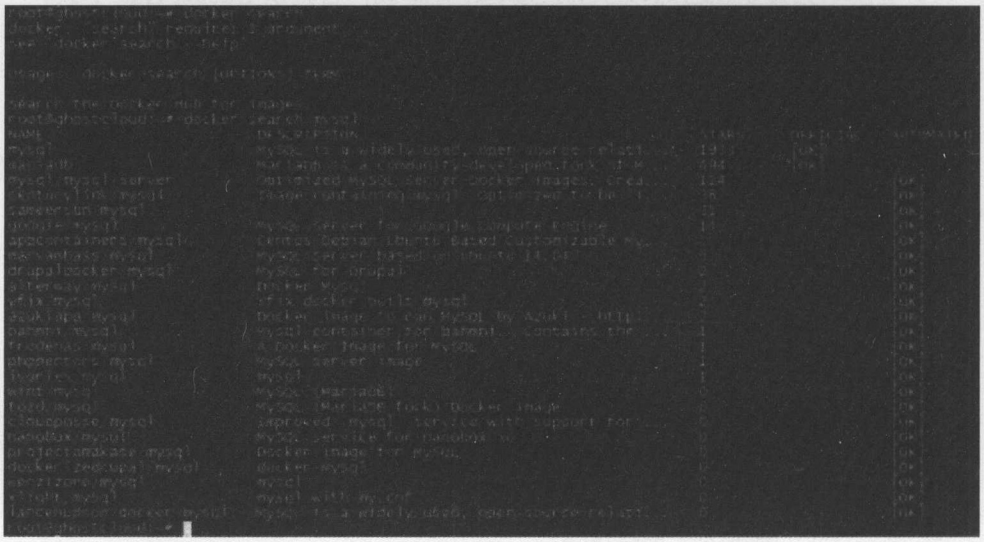


图 4.5　查找 Docker Hub 镜像

由于 Docker Hub 在国外，所以有时候搜索会失败，可以多次尝试。

4.5.4　查找其他仓库镜像

如果不是其他仓库的镜像，目前只能通过 restful 接口查找。例如，搜索 hub.ghostcloud.cn 的镜像，可以用以下命令。

```
root@ghostcloud:~# curl http://hub.ghostcloud.cn/v2/_catalog
{"repositories":
    ["alpine",
      "apache-php",
      "apache_tomcat",
      "apachetomcat",
      "busybox",
      "centos",
      "chenpeng/12300000",
      "chenpeng/chenpengtestimage",
      "chenpeng/java",
      "chenpeng/jave",
      "chenpeng/mmmmmmmmm",
      "chenpeng/testimage",
      "debian",
      "discuz",
```

```
"discuz-zh",
"django",
"drown_scan",
"ecshop",
"fedora",
"gcauth",
"gcbuild",
"gchelp/ghostcloud",
"gchomepage",
"gcmongodb",
"gcnginx",
"gcplatform",
"gcrelease",
"gcrepimg",
"gcserver",
"gctestlink",
"gitlab-ce",
"golang",
"hadoop-docker",
"httpd",
"httpd-image-php5",
"java",
"lamp",
"library/alpine",
"library/buildpack-deps",
"library/busybox",
"library/cassandra",
"library/centos",
"library/debian",
"library/django",
"library/docker",
"library/elasticsearch",
"library/fedora",
"library/ghost",
"library/golang",
"library/haproxy",
"library/hello-world",
"library/httpd",
"library/iojs",
"library/java",
"library/jenkins",
"library/jetty",
"library/jruby",
"library/kibana",
"library/logstash",
"library/mariadb",
"library/maven",
"library/memcached",
"library/mongo",
"library/mysql",
"library/neo4j",
```

```
        "library/nginx",
        "library/node",
        "library/oraclelinux",
        "library/owncloud",
        "library/php",
        "library/postgres",
        "library/python",
        "library/rabbitmq",
        "library/rails",
        "library/redis",
        "library/registry",
        "library/rethinkdb",
        "library/ruby",
        "library/swarm",
        "library/tomcat",
        "library/ubuntu",
        "library/ubuntu-debootstrap",
        "library/wordpress",
        "liufm/ping",
        "liufm/ttwwe",
        "liuj/test",
        "lnmp",
        "maxiaoping/test1image",
        "mongo",
        "mongodb",
        "mysql",
        "nodejs",
        "php",
        "php5.6-apache",
        "php5.6-cli",
        "php5.6-fpm",
        "php5.6-zts",
        "phpmyadmin",
        "postgres",
        "python"
    ]
}
```

如果需要查看 Ubuntu 的版本，可以用以下链接。

```
root@ghostcloud:~# curl http://hub.ghostcloud.cn/v2/ubuntu/tags/list
{"name":"ubuntu","tags":["14.04"]}
```

4.5.5　push 镜像

之前都是从 Hub 上拉取镜像，本节将讲解如何向 Hub 上 push 镜像。在 push 之前需要首先 tag 镜像，如图 4.6 所示。

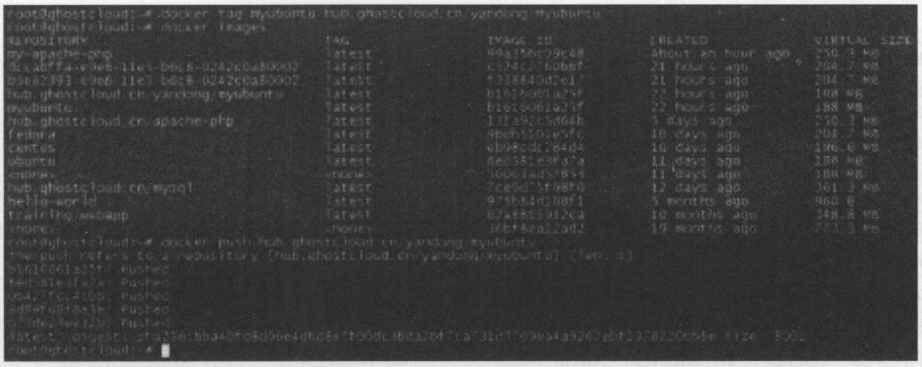


图 4.6　tag 镜像

tag 的语法如下，它可以把一个镜像的 push 目的地进行标记，为 push 做准备。将本地的 myubuntu 标记为 hub.ghostcloud.cn/yandong/myubuntu，之后再调用 push 到远端的私有仓库。

```
root @ ghostcloud:~# docker tag
docker: "tag" requires 2 arguments.
see 'docker tag --help!

usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]
Tag an image into a repository
root@ghostcloud: ~#
```

4.5.6　根据 Dockerfile 编译镜像

如果对 Linux 开源软件熟悉，则一定知道 Makefile。Dockerfile 和 Makefile 类似，也是一个编译脚本，用于生成镜像。

（1）创建一个文件夹，并新建一个 Dockerfile。

```
root@ghostcloud:~/myimage# cat Dockerfile
# This is a comment
FROM ubuntu:latest
MAINTAINER Shev Yan <yandong_8212@163.com>
CMD echo 'hello my image from Dockerfile.'
```

（2）build 命令。

```
root@ghostcloud:~/myimage# docker build .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:latest
 ---> 8ed581e3fa7a
Step 2 : MAINTAINER Shev Yan <yandong_8212@163.com>
 ---> Running in 9b9d552f8184
 ---> 727c430beada
Removing intermediate container 9b9d552f8184
Step 3 : CMD echo 'hello my image from Dockerfile.'
 ---> Running in 343f2abeb557
```

```
 ---> 30331f9ff01a
Removing intermediate container 343f2abeb557
Successfully built 30331f9ff01a
```

（3）查看并运行新生成的镜像。

```
root@ghostcloud:~/myimage# docker images myimage
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
myimage         latest          30331f9ff01a    4 minutes ago   188 MB
root@ghostcloud:~/myimage# docker run --rm myimage
hello my image from Dockerfile.
```

4.5.7　删除镜像

删除镜像是通过 docker rmi 来进行的，删除之前要确保没有容器在使用该镜像，否则会报错。

```
root@ghostcloud:~/myimage# docker rmi myimage
Untagged: myimage:latest
Deleted: 30331f9ff01ae83a3eea81782e1f3083d63803866e3cecd5bdca54a92cafa9c6
Deleted: 727c430beada158f26a0a7c19caa864a66e8330354ef0a022b0a8ca08fe6f0ba
```

4.6　docker run 命令

docker run 是用得最多，也是最为复杂的命令，可以通过它来启动一个容器。本节将对 run 的常用参数做介绍。Docker 的更新速度非常快，本书以 Docker 1.10 作为讲解的版本。

4.6.1　docker run 的语法格式

docker run 的基本格式如下：

```
$ docker run [OPTIONS]IMAGE[:TAG|@DIGEST][COMMAND][ARG...]
```

这个命令必须指定 IMAGE，一个镜像默认包含了以下信息：

- 前台运行还是后台运行；
- 容器的 ID；
- 网络设置；
- 运行中的 CPU 和内存。

但是几乎所有的信息，都可以通过 docker run [OPTIONS]来进行修改。OPTIONS 是一个可选项，如果不选择该选项，就会首先使用镜像的参数，然后再使用全局的默认参数。所有的以下介绍都可以通过 man docker run 得到。

说明　在命令行程序里面，通常使用中括号表示该部分为可选项，使用尖括号表示该部分是必须有的参数。

4.6.2　前后台运行

1. 后台运行（-d）

在启动容器时，如果要将容器放在后台运行，则可以使用-d=true 或者-d 参数，-d 的英文对应的是 detached，其原理就是不与容器的 stdin、stdout 进行绑定。有些时候，可能使用 service xxx start 命令作为容器启动命令，但这是有问题的。虽然这个命令执行成功了，但是后续就退出了，紧接着容器也会退出。因此，容器的生存周期是直接和启动容器的命令生命周期一致的，虽然可以通过该命令 fork 出子进程，但是一旦主进程退出，整个容器就结束了。如果需要在容器中运行多个程序，则可以使用 Supervisord。

注意　当以-d 参数运行后，如果要再次进入容器，则可以使用 docker attach <cid> 的方式重新绑定到当前 shell 的终端上。如果要再次进入-d 模式，不能输入【ctrl+c】，而应该使用【ctrl+pq】。

2. 前台运行

如果不选择-d 参数，则表示放在前台运行。有些时候，若要 attach 到容器里面的 shell，则可以通过-i -t 参数。

```
root@ghostcloud:~# docker run -it ubuntu /bin/bash
root@cb90c945e2b8:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
```

注意　当进入容器时，默认的主机名就是容器的短 ID - cb90c945e2b8。同时，在涉及容器和镜像操作时，都可以仅输入前面的 3～4 个 ID 字符即可，Docker Daemon1 会自动做匹配。

4.6.3　容器的标识

1. --name 属性

容器有三种方式进行标识，见表 4.1。

表 4.1　容器标识方式

名　称	举　例
长 UUID	f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778
短 UUID	f78375b1c487
Name	evil_ptolemy

UUID 是 Docker daemon 产生的，在一台主机上是唯一的。在创建容器时，可以通过--name 指定容器的名字，如果不指定则系统自动分配一个字符串作为名称。该 name 一个比较大的用途就是，可以用于在两个容器之间建立 link 通信。

2. Image[:tag]

之前已经讲过，docker run 时必须指定镜像名，这个就是它的格式，例如：

```
$ docker run ubuntu:14.04
```

3. Image[@digest]

从镜像的 V2 版本格式开始，每个镜像都包含了一个含有镜像内容的签名，用户同样也可以通过这种方式来指定镜像。通过 docker insepect <image_id>可以查看到 digest。

4.6.4 PID 设置

PID 用于控制容器中的进程使用什么 pid。一般来说，主机上的进程 ID 是从 1 开始的，通常是 init 进程，而容器中执行的程序的 pid 是也从 1 开始的，这就是利用 pid namespace 实现的。

```
--pid="": Set the PID (Process) Namespace mode forthe container, 'host': use the
host's PID namespace inside the container
root@cb90c945e2b8:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 02:00 ?        00:00:00 /bin/bash
root        16     1  0 03:47 ?        00:00:00 ps -ef
```

如果需要在容器里面共享主机的 PID Namespace，那就加上--pid=host，之后的情况如下。

```
root@ghostcloud:~# docker run -it --rm --pid=host ubuntu /bin/bash
root@158c0108a34f:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Mar12 ?        00:00:03 /sbin/init
root         2     0  0 Mar12 ?        00:00:00 [kthreadd]
root         3     2  0 Mar12 ?        00:00:08 [ksoftirqd/0]
root         5     2  0 Mar12 ?        00:00:00 [kworker/0:0H]
root         7     2  0 Mar12 ?        00:00:13 [rcu_sched]
...
```

该容器和主机共用 PID。

4.6.5 UTS(--uts) 设置

Docker 1.10 之后的版本支持以下设置：

```
--uts="": Set the UTS namespace mode forthe container, 'host': usethe host's UTS
namespace inside the container
```

UTS 可以使容器同主机使用相同的 hostname 和 domain，使用时要慎重。

4.6.6　IPC(--ipc)设置

IPC 是进程间通信的支持，可以和主机共享。

```
--ipc="": Set the IPC mode forthe container, 'container:<name|id>': reuses another
container's IPC namespace 'host': use the host's IPC namespace inside the container
```

4.6.7　网络设置

容器有五种网络方式，默认使用 bridge，通过主机和容器的端口映射通信。

```
--dns=[] : Set custom dns servers for the container
--net="bridge" : Connect a container to a network
'bridge': createa network stack onthe defaultDocker bridge
'none': nonetworking
'container:<name|id>': reuseanother container's network stack
'host': use the Docker host network stack
'<network-name>|<network-id>': connect to a user-defined network
--net-alias=[] : Add network-scoped alias for the container
--add-host="" : Add a line to /etc/hosts (host:IP)
--mac-address="" : Sets the container's Ethernet device's MAC address
--ip="" : Sets the container's Ethernet device's IPv4 address
--ip6="" : Sets the container's Ethernet device's IPv6 address
```

同时，也可以给容器设置 dns。表 4.2 是 Docker 支持的各种网络模式。

表 4.2　Docker 支持的网络模式

网　络	描　述
none	不在容器中使用网络
bridge（默认）	容器通过 veth 连接到主机的桥接上
host	使用和主机相同的网络
container:<name|id>	使用其他容器的网络
network	使用用户自定义的网络（通过 docker network create 创建）

1. none

容器不能访问外部网络，内部存在回路地址。

2. bridge

桥接是在主机上的，通常为 docker0，每启动一个容器时，会为该容器创建一个 veth，veth 一端连接到 docker0，另一端连接到容器内的 eth0，如图 4.7 所示。

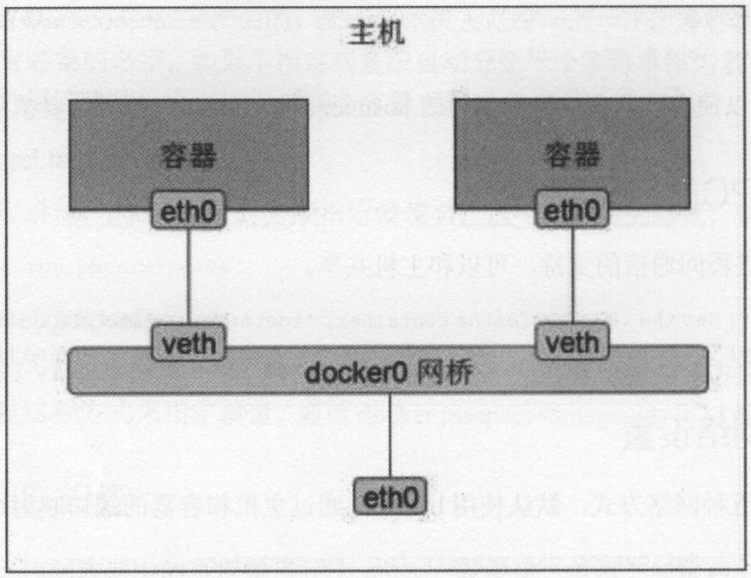


图 4.7　桥接网络模型

3．host

host 就是使用主机的网络，并不特殊。相比于桥接，该模式由于没有通过 iptable 的转发，性能上要比桥接好一些。但是根据测试，也只有 15%左右的损失，相比于 nat 的封闭和安全性，更建议使用桥接。在对网络性能特别关注时，如 HAProxy 等负载均衡器时，可以使用 host 模式。

4．container

该模式将容器的网络栈合并到了一起，如果二者在同一台机器，且不需要独立的网络地址，是比较好的一种选择。

5．自定义网络

network 是用户自定义的网络，这一功能主要是为了解决 Docker 跨网络通信能力不足的问题和特殊网络需求问题。对于 Overlay 和用户定义的 multi-host 模式，就应该使用该模式。

6．/etc/hosts 管理

通过--add-host 可以在容器中添加 ip 到 host 的解析规则。

4.6.8　重启策略（--restart）

--restart 参数用来指定当容器退出后的行为，当容器重启时，docker ps 可以看到处于 Up 或 Restarting，也可以通过 docker events 命令查看相关信息。目前，Docker 支持多种 restart 策略，见表 4.3 所示。

表 4.3　容器重启模式

策　略	作　用
No	没有任何重启操作，也是默认的属性
On-failure	当容器的命令反回非 0 值，即有错时，进行重启
Always	无论容器处于什么状态，都重启。同时，容器会在 daemon 启动时，附带自启动
Unless-stopped	与 Always 类似，但不会随 daemon 自启动

容器的退出状态就是执行命令的错误码，Linux 下通常用 0 表示正常，其他表示错误。

4.6.9　Clean up (--rm)

默认情况下容器退出后，并不会被删除，如果通过 docker ps -a，可以看到所有的容器，包括运行中的和停止的。通过--rm 可以在容器退出时自动删除容器，方便清楚残留的文件。

4.6.10　CGroups 控制

CGroups 是用来限定容器中资源使用情况的。CGroups 具体可以控制的东西如下：

- 用户态内存的控制；
- 内核态内存控制；
- CPU 的控制；
- 磁盘 I/O 的控制；
- 设备读写速率的控制；
- 内存耗尽（OOM）时的行为。

具体的细节可以查看：

https://docs.docker.com/engine/reference/run/#specifying-custom-cgroups

4.6.11　特权模式和 Capabilities

默认情况下，容器是运行在非特权模式下的，这种情况下，从容器内是不能访问任何宿主机的设备的。但是有些时候，用户可能需要访问某些设备，就必须使用--privileged 参数。如果希望使用某个特定设备，则可以使用--device，如你想使用 GPU 时：

```
$ docker run --device=/dev/snd:/dev/snd...
```

默认情况设备是可读可写的，并且支持 mknod，也可以通过:rwm 来修改。

```
$ docker run --device=/dev/sda:/dev/xvdc--rm -it ubuntu fdisk /dev/xvdc
```

```
Command (m forhelp):q

$ docker run --device=/dev/sda:/dev/xvdc:r--rm -it ubuntu fdisk /dev/xvdc
You will notbe able to write the partition table.

Command (m forhelp):q

$ docker run --device=/dev/sda:/dev/xvdc:w--rm -it ubuntu fdisk /dev/xvdc crash....
 $ docker run --device=/dev/sda:/dev/xvdc:m--rm -it ubuntu fdisk /dev/xvdc
fdisk:unable to open /dev/xvdc:Operation notpermitted
```

除此之外，还可以使用--cap-add 或--cap-drop 设置 Capabilities。

4.6.12　日志驱动（--log-driver）

容器默认的日志是输出到 stdout, stderr 中的，也可以在 Docker Daemon 中设置不同的日志输出方式。通过--log-driver=VALUE 也能够为单个容器指定不同的日志格式，容器日志驱动方式见表 4.4。

表 4.4　容器日志驱动方式

日 志 驱 动	描　　述
none	不显示日志，Docker logs 没有输出
json-file	默认的日志输出方式，通过 JSON 来保存日志
syslog	将日志输出到系统日志中，通常是/var/log/message
gelf	Graylog Extended Log Format(GELF)日志，日志会写入到 GELF 的收集器中，如 Graylog 或 Logstash
journald	将日志写入到 journald
fluentd	写入到 fluentd
awslogs	写入到 Amazon CloudWatch
splunk	写入到 splunk 日志收集器中

4.6.13　覆盖 image 的默认参数

当用户编写Dockerfile时可能会带一些默认参数，目前只有FROM, MAINTAINER, RUN 和 ADD 是不能覆盖的，其他都是可以覆盖的。例如：

- CMD——默认命令或属性；
- ENTRYPOINT——默认执行的命令；
- EXPOSE——导出的端口；
- ENV——环境变量；
- VOLUME——卷挂载；
- USER；
- WORKDIR。

说明　CMD 和 ENTRYPOINT 的区别是，CMD 其实更像是一个命令参数，若在 run 的时候指定了命令，然后 CMD 就会作为参数；ENTRYPOINT 是镜像的默认启动项，所以 CMD 和用户所指定的参数都会作为 ENTRYPOINT 的参数。

4.7　习题

本章介绍了 Docker 的原理，以及镜像制作过程，并深入介绍了 run 命令。接下来，通过习题和实验检验本章的学习成果。

（1）搭建一个 Web 服务器，并通过 8080 端口导出服务。

（2）docker exec 和 docker attach 的区别是什么？

（3）通过 docker exec 从基础 Ubuntu 镜像，制作一个 Apache 镜像。

（4）通过 Dockerfile 制作一个 Apache 镜像。

（5）Docker 有哪几种网络模式，各有什么特点？

第5章　容器的网络

Docker 最初的网络是比较单一的，功能也相对偏弱，随着 1.9 版本的推出，其网络部分得到了很大的提升。在本章将对容器的网络做一个简单的介绍，包括：

- 容器自带网络；
- 网络详情；
- 用户自定义网络。

5.1　容器自带网络

当 Docker 成功安装后，就会创建三种网络，可以通过 docker network ls 进行查看：

```
$ docker network ls
NETWORK ID NAME DRIVER
7fca4eb8c647 bridge bridge
9f904ee27bf5 none null
cf03ee007fb4 host host
```

这三种网络就是系统自带的，创建容器时可以通过--net 进行指定。对于 bridge 而言，默认是主机挂接在主机的 docker0 上的，在主机上通过 ifconfig 可以查看到：

```
root@ghostcloud:~# ifconfig
br-40719032dd42 Link encap:Ethernet  HWaddr 02:42:02:e6:b2:db
          inet addr:172.18.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:2ff:fee6:b2db/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:76 (76.0 B)  TX bytes:168 (168.0 B)

docker0   Link encap:Ethernet  HWaddr 02:42:8c:ca:18:18
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
```

```
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 00:0c:29:0f:5d:4d
          inet addr:192.168.10.10  Bcast:192.168.255.255  Mask:255.255.0.0
          inet6 addr: fe80::20c:29ff:fe0f:5d4d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:368767 errors:1 dropped:181 overruns:0 frame:0
          TX packets:25936 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:68266498 (68.2 MB)  TX bytes:31563732 (31.5 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

5.2　网络详情

可以通过 docker network inspect <net>查看本机 Docker 的网络信息。

```
root@ghostcloud:~# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "33e251770bda7a9adfbc0ef7b7f468a71f577cc8af0e1a44369a5a295fdc66b0",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
```

```
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

当启动一个容器时，在 Containers 下面加入 Subnet 和 Gateway。

```
root@ghostcloud:~# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "33e251770bda7a9adfbc0ef7b7f468a71f577cc8af0e1a44369a5a295fdc66b0",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {
            "7849f9e530c7d650646673e64755d933c56051326a9e57586a8a272c415a2b0d": {
                "Name": "elated_dijkstra",
                "EndpointID":
"0730a4057116c296a1c0c19c796d68cb6f949284c8777c42094ee91a04b04f5c",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

每运行一个容器，都会在全局注册相关的网络信息。

5.3　用户自定义网络

除默认的网络外，Docker 还允许用户创建自己的网络，主要包括三种：桥接网络；Overlay 网络；插件网络。

5.3.1　桥接网络

系统默认的桥接是 docker0，若要将多个容器隔离在一个新的桥接网络中，则可以使用以下命令。

```
root@ghostcloud:~# docker network create --driver bridge mynet
e4a0a19ebb624f82b396915a6439f4bf4c8d520f50fb29ddc86dfd012106c6d6

root@ghostcloud:~# docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id": "e4a0a19ebb624f82b396915a6439f4bf4c8d520f50fb29ddc86dfd012106c6d6",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]
```

可以通过--net 属性将容器挂接到 mynet 中。

```
root@ghostcloud:~# docker run --net=mynet --rm -it ubuntu

root@ghostcloud:~# docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id": "e4a0a19ebb624f82b396915a6439f4bf4c8d520f50fb29ddc86dfd012106c6d6",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
```

```
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1/16"
                }
            ]
        },
        "Containers": {
            "b8321c436612a350964ad08c0a1f6c31da328c67e399b30be160eefe1a9a0631": {
                "Name": "desperate_hodgkin",
                "EndpointID":
"14d0e3aa846bba610b19eaca31436d4b80e190796a9b2f1f45e7731fa0110575",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

在同一个桥接下，形成了一个私网，相互间是可以通信的，但是这仅限于在同一台主机上。若要跨主机通信，就必须使用 Overlay 网络。

5.3.2　Overlay 网络

Overlay 是一种虚拟交换技术，主要是解决不同 IP 地址段之间的网络通信问题。Docker 使用的 Overlay 技术是 VXLAN，是借助于 libnetwork 实现的。Overlay 网络需要一个 K-V 服务来存储相关的主机信息。目前，Docker 支持的 K-V 存储服务有 Consul、Etcd 和 ZooKeeper。其中，Consul 和 Etcd 都是用 Golang 来进行开发的，ZooKeeper 是用 Java 开发的。由于 Golang 的原生跨平台型，Consul 就成为了默认的发现服务。

对于 Overlay，主机还必须开放 UDP/4789 和 TCP/UDP/7946，分别用作数据通道和控制通道。以下就来创建一个 Overlay 网络（如图 5.1 所示）。

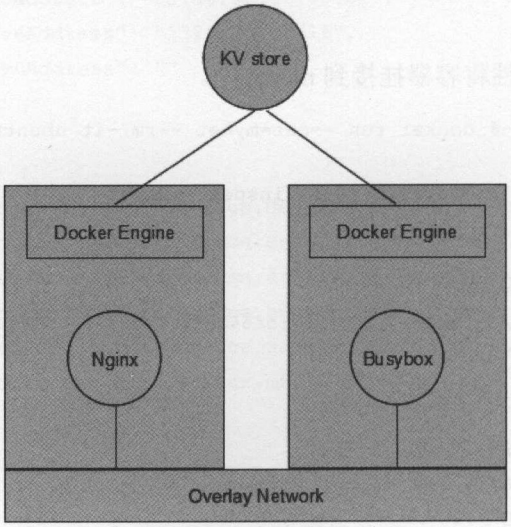


图 5.1　Overlay 网络模式

圆形的部分都是容器，步骤如下。

（1）创建一个 Docker Machine 用于运行 KV store。

```
$ docker-machine create -d virtualbox mh-keystore
```

说明　本节演示的例子，需要在主机上安装 Docker-machine 工具，这样可以方便测试。

（2）在#1 中创建的虚拟机中，运行 consul KV store。

```
$ docker $(docker-machine config mh-keystore) run -d -p "8500:8500" -h consul
progrium/consul -server -bootstrap
#查看运行状态
$ docker $(docker-machine config mh-keystore) ps
CONTAINER ID     IMAGE            COMMAND                  CREATED
STATUS           PORTS
NAMES
8c16fceaeeff        progrium/consul     "/bin/start -server -"   2 minutes ago
Up 2 minutes         53/tcp, 53/udp, 8300-8302/tcp, 8400/tcp, 8301-8302/udp,
0.0.0.0:8500->8500/tcp   tender_golick
```

说明　consul 是一个 KV 的中间件，由 Golang 编写，主要用来做服务、发现和注册。

（3）创建两个 VM，其中一个作为 swarm 的 master。

```
$ docker-machine create \ -d virtualbox \ --swarm --swarm-master \
--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500"\
--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore):8500"\
--engine-opt="cluster-advertise=eth1:2376"\ mhs-demo0

$ docker-machine create -dvirtualbox \ --swarm \ --swarm-discovery="consul:
//$(docker-machine ip mh-keystore):8500"\ --engine-opt="cluster-store=consul://$(docker-
machine ip mh-keystore):8500"\
--engine-opt="cluster-advertise=eth1:2376"\ mhs-demo1
```

（4）创建一个 Overlay 网络。

```
$ docker $(docker-machine config mhs-demo0) network create -d overlay my-net
26c7f5d3fff8e3cb725c93c486da2ff9a8efbea19a90859e00376c9ca08f622a

Administrator@EKEUSER-PC MINGW64 /
$ docker $(docker-machine config mhs-demo0) network ls
NETWORK ID          NAME                DRIVER
26c7f5d3fff8        my-net              overlay
33a92eabdd52        bridge              bridge
32c656ab911d        none                null
75cec8a41354        host                host

Administrator@EKEUSER-PC MINGW64 /
```

```
$ docker $(docker-machine config mhs-demo1) network ls
NETWORK ID          NAME                DRIVER
26c7f5d3fff8        my-net              overlay
49eb74d5a193        bridge              bridge
a5483ee39c78        none                null
95e713841aba        host                host
```

说明　Docker swarm 是 Docker 的另一个开源项目，主要用来做集群管理。

（5）在一台主机上创建一个 Nginx 容器。

```
$ docker $(docker-machine config mhs-demo1) run -itd --name=web --net=my-net nginx
```

说明　Nginx 是一个开源的 Web 容器和反向代理项目，在互联网行业中使用非常普遍。

（6）在另一台机器上访问 Nginx。

```
$ docker $(docker-machine config mhs-demo0) run --rm --net=my-net busybox wget
-O- http://web
Connecting to web (10.0.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
-                    100% |*******************************|   612  0:00:00 ETA
```

注意　本例通过-rm 让容器运行结束后就自动销毁，这种方式也称为工具型容器。用户可以将一些工具的程序和环境打包到镜像中，需要时启动容器即可。

5.4　习题

本章详细介绍了 Docker 的三种网络模式，并介绍了通过 swarm 构建 Overlay 网络。接下来，通过习题和实验检验本章的学习成果。

（1）自建一个桥接网络，并在该网络下关联两个容器，之后相互通信。

（2）根据 5.3 节的例子，自己在你的机器上通过 Docker Machine 和 docker-swarm 构建一个 Overlay 网络，并在网络上创建两个容器进行相互间通信。

第6章　容器的数据

容器中的文件系统是由分层文件系统提供的，包含只读层（镜像）和可读可写层（容器运行时层），这些都是被封装在容器内部的。如果用户需要将主机上的文件系统共享给容器使用，那怎么办呢？目前有两种处理方式：

- 数据卷——将主机的卷 mount 进入容器；
- 数据容器——将外部容器分享给容器。

6.1　数据卷

数据卷提供了一种主机和容器共享数据的方式，有些时候需要用它来做持久化和数据共享。当做持久化时，通常数据卷都会比较大，可以将其放在单独的磁盘、卷或者阵列上，这个时候容器只是一个执行环境。当做数据共享时，可以用于开发和测试分布式系统，如需要用到共享盘、处理 fencing 等。数据卷主要通过-v 参数来指定。

6.1.1　创建一个数据卷

若需要一个外部卷来存放持久化数据，而不想把数据包含在容器内部。例如：

```
root@ghostcloud:~# docker run -d -P --name datatest -v /webapp ubuntu
13ffee3a3f50d07fd5a737aaf2efc60ceec28e2b04c5d534cfd84d8b70019c11
```

这条命令创建了一个名为 datatest 的容器，同时为其创建了一个/webapp 的数据卷，这是数据卷在其内部的位置。那么它在主机上的什么位置呢？

```
root@ghostcloud:~# docker inspect 13ff

...
"Mounts": [
          {
              "Name":
"ec1c427a6a76be4918d6e8bac3247e2836dc8f424c9e06466fcf1baab-6e7ee79","Source":
"/var/lib/docker/volumes/ec1c427a6a76be4918d6e8bac3247e2836dc8f424c9e06466fcf
```

```
1baab6e7ee79/_data",
                "Destination": "/webapp",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ],
    ...
```

它的位置在/var/lib/docker 下。

6.1.2　映射一个外部卷

如果以-v src:des 的方式指定，那么容器则会直接将宿主机的目录挂载到容器内部：

```
root@ghostcloud:~# docker run -it -v /root:/hostroot ubuntu
root@86ea1246ae5d:/# ls /hostroot
Dockerfile  composetest  gcagent  myimage  uninstall_agent.sh
```

将宿主机/root 目录映射到了容器/hostroot 目录中。此时，用户可以在容器中对宿主机/root 目录中的文件进行修改，但这种操作是非常危险的。在做数据卷映射时，一定要特别小心，任何时候都不要将宿主机的根目录映射到容器内部。

6.2　使用数据型容器

由于容器本身就可以包含文件系统，那么可不可以把容器的卷分享给另一个容器用呢？答案是可以的。具体的步骤如下。

（1）创建一个包含外部卷的容器，注意是 create，并不是 run。run 是 create 后再 start，本例只需要容器的文件系统，所以只需要 create.

```
root@ghostcloud:~# docker create -v /dbdata --name dbstore ubuntu
d95cdc1139ed1011fe51843f524c377cd7497629e9a4434508f422f15b61a03c
```

（2）在另一个容器中通过--volumes-from 来映射。

```
root@ghostcloud:~# docker run --rm -it --volumes-from dbstore ubuntu
root@4b61bb181471:/# df -h
Filesystem                                      Size  Used Avail Use% Mounted on
none                                            8.8G  4.2G  4.2G  50% /
tmpfs                                           2.0G     0  2.0G   0% /dev
tmpfs                                           2.0G     0  2.0G   0% /sys/fs/cgroup
/dev/disk/by-uuid/27d8b1c5-4bfc-4499-94d6-6e5f5c42e923  8.8G  4.2G  4.2G  50%
/dbdata
```

6.3　备份、还原和迁移数据卷

下面是通过容器型数据卷和数据卷联合使用做备份的例子。

```
$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf
/backup/backup.tar /dbdata
```

剖析：

- --volumes-from 表示使用 dbstore 这个容器的数据卷；
- -v $(pwd):/backup 表示将当前路径映射到容器的/backup 中，用于后续备份；
- ubuntu tar cvf /backup/backup.tar /dbdata 表示将/dbdata 的内容备份到当前目录。

上面这个例子就是典型的将容器作为一个工具来使用的例子，如果更进一步，用户可以自己写一个 Dockerfile，然后产生一个 Image，将参数都指定好，以后只需启动容器就可备份，备份完成后又自动退出。

那么还原呢？

```
$ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd
/dbdata && tar xvf /backup/backup.tar --strip 1"
```

类似地，通过 tar 来解压即可。

6.4　容器和代码进行关联

数据卷有下面几个特点：

- 数据卷在容器创建时进行初始化；
- 数据卷既可以共享，也可以在容器之间重用；
- 对于数据卷的读写是直接下发的；
- Commit 命令不会将改动保存到镜像中；
- 即使容器被删除了，数据卷仍然存在，因此这一块需要特别注意，避免产生垃圾数据卷。

6.5　习题

本章介绍了在容器中通过数据卷和容器卷保存需要持久化的数据。接下来，通过习题和实验检验本章的学习成果。

（1）在主机上新建 index.html 文件，内容为“ghostcloud”。启动一个 httpd 容器，通过挂载卷的方式，把主机上的 index.html 文件挂载到容器中。通过浏览器访问 http://ip:port/index.html，在浏览器中看见“ghostcloud”。

（2）新建一个 Ubuntu 容器，名为 datastore，作为数据卷容器。在该容器中，新建 index.html 文件，内容为“ghostcloud”。启动一个 httpd 容器，挂载 datastore 数据卷容器。通过浏览器访问 http://ip:port/index.html，在浏览器中看见“ghostcloud”。

第7章 镜像仓库

从 hello-world 开始，就一直在和镜像仓库打交道，而且也许会经历拉取镜像失败的情况，这主要归咎于国内网路的一些限制。镜像仓库顾名思义就是存放镜像的，有了它就可以在不同的主机上使用相同的镜像，有了它就可以使私有云和公有云进行联通。一般来说，可能会用到下面几种仓库。

- Docker Hub 主仓库：这是所有的公共仓库，截至 2016 年 1 月，已经有超过 10 亿次的下载量。Docker Hub 有着最全面的镜像，也是 Docker 客户端默认的拉取仓库，但是由于其在国外，拉取和查询速度都非常慢。而且它默认镜像是全公开的，即镜像可以被任何人下载。
- 内部私有仓库：用户可以在内部创建私有的仓库。
- 公共的私有仓库：将仓库放在公网上，并自己管理权限。

7.1 仓库相关的 Docker 命令

镜像仓库用于保存镜像，用户可以从镜像仓库中下载镜像、上传镜像、修改镜像。镜像的主要操作包含 search、pull、login 和 push。

7.1.1 登录

有些镜像仓库是需要认证的，从这些镜像仓库搜索镜像、下载镜像、上传镜像都需要首先登录。

```
root@ghostcloud:~# docker login --help

Usage:  docker login [OPTIONS] [SERVER]

Register or log in to a Docker registry.
If no server is specified, the default is defined by the daemon.

  -e, --email       Email
  --help            Print usage
```

```
-p, --password    Password
-u, --username    Username
```

如果选择 SERVER 将会使用 Docker Hub，否则可以指定一个第三方 SERVER。当用户登录成功后，用户的认证信息是保存在~/.doker/config.json 中。

7.1.2　查找

之前其实我们已经使用过查找这个命令了，这里再来查找一下与MySQL相关的镜像。

```
root@ghostcloud:~# docker search mysql
NAME                DESCRIPTION                    STARS   OFFICIAL   AUTOMATED
mysql               MySQL is a widely usedi...     1948    [OK]
mysql/mysql-server  Optimized MySQ...              116                [OK]
centurylink/mysql   Image containing my...         39                 [OK]
sameersbn/mysql                                    31                 [OK]
google/mysql        MySQL server for...            14                 [OK]
......
```

7.1.3　拉取

使用下面的命令拉取 MySQL 镜像：

```
root@ghostcloud:~# docker pull mysql
```

7.1.4　提交

提交就是通过 push 来完成的。在 push 之前一般都需要首先 login，然后再提交到指定的目录位置，其格式为：

```
$ docker push  [OPTIONS] [server/][user/]imagename[:TAG]
```

最后的那部分是通过 docker tag 来标记的，也就是说，如果要提交到某个地方，首先是要 docker tag 成指定镜像名，然后再进行 push。

7.2　习题

本章介绍了镜像仓库相关操作。接下来，通过习题和实验检验本章的学习成果。

（1）在 www.dockerhub.com 上注册一个账户。在命令行中，通过 docker login 命令登录。

（2）在命令行中，通过 docker search 命令搜索 golang 镜像。

（3）在命令行中，通过 docker pull 命令下载 golang 镜像。

（4）在命令行中，通过 docker tag 命令给 golang 镜像打标签。标签格式为 username/golang:test。通过 docker push 命令提交镜像。

第 3 篇

Docker 的高级使用

第 8 章　镜像和容器的存储结构

镜像和容器在 Docker 中是两个非常重要的概念。很多初学者不清楚两者之间的关系，很多时候会把两者混淆。简单地讲，镜像是一些程序和文件的集合，容器是镜像的一个运行实例，两者有着紧密的联系。

Docker 为了节约存储空间及共享数据，会对镜像和容器分层。不同镜像可以共享相同数据，例如，从同一个基础镜像中生成的子镜像可以共享基础镜像，从同一个镜像启动的不同容器也可以共享这个镜像。

Docker 为了加快容器的启动速度，在启动容器时，会在镜像上为容器分配一个可写数据层，在容器运行中，新增、修改和删除的数据都保存在这个容器层中。

Docker 支持多种不同的存储方式，每种存储方式保存镜像和容器的方法都不相同。下面将具体介绍这些区别。

8.1　镜像、容器和存储驱动的关系

为了更好地使用存储驱动，首先需要了解 Docker 是如何编译和存储镜像的。接下来，需要知道容器启动时，如何使用镜像。最后，需要了解镜像和容器使用的存储技术。

8.1.1　镜像和镜像层

每个镜像都由多个镜像层组成。这些镜像层都是只读的，从下往上，以栈的方式组合在一起，组成容器的根文件系统。图 8.1 所示是 Ubuntu 15.04 的镜像结构图，该镜像由四个镜像层组成。

Docker 的存储驱动用于管理这些镜像层，对外提供一个单一的文件系统。

镜像中的层都是只读的。镜像启动容器后，在容器中需要有可读写的区域，用于

存储容器运行时的数据。当启动容器时，Docker 会新建一个可读写的容器层，把这个容器层添加在镜像层上。这个容器层是 thin 类型的层，采用预分配存储空间的方式。在开始时并不分配存储空间，当需要新建文件或修改文件时，才从存储池中分配一部分存储空间。容器运行时，所有文件变化的数据都保存在容器层中，如新建文件、修改文件、删除文件。

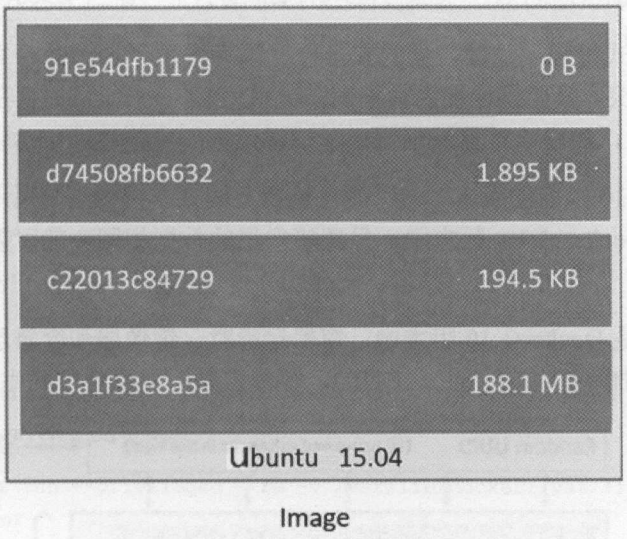


图 8.1　Ubuntu 15.04 的镜像结构

在图 8.2 所示中，使用 Ubuntu 15.04 镜像启动一个容器。容器的存储空间由两部分组成，底部是镜像层，顶部是容器层。镜像层是只读层，容器层是可读写层。容器运行时，涉及文件写的操作，都是在容器层中完成。

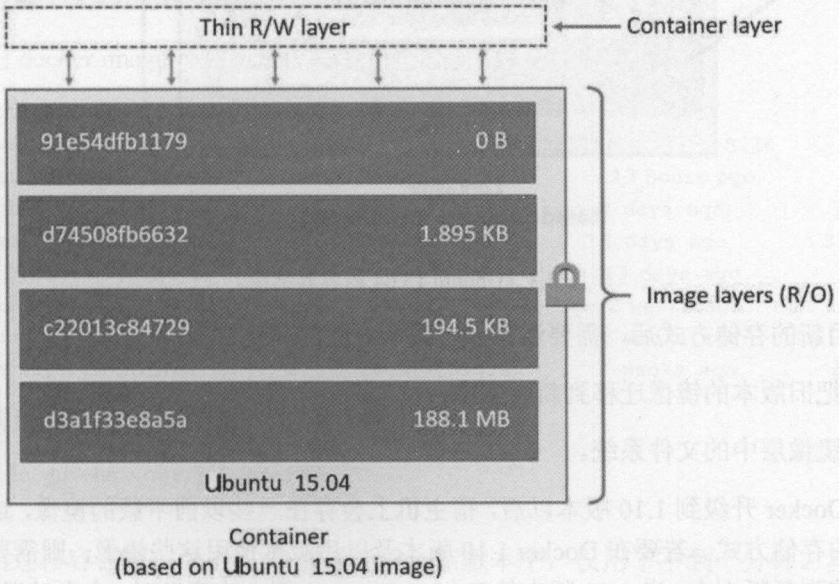


图 8.2　Ubuntu 15.04 的容器结构

8.1.2　镜像存储方式

Docker 1.10 版本采用了一种全新的镜像存储方式。为了区别镜像层，Docker 为每个镜像层计算了一个 UUID。在 Docker 1.10 以前的版本中，会根据镜像层中的数据产生一个随机码，用随机码作为镜像层的 UUID。在 Docker 1.10 版本中，会根据镜像层中的数据，使用加密哈希算法生成镜像层的 UUID。

这种新的方式增强了镜像数据的安全性。因为采用了加密哈希算法，所以避免了镜像层 UUID 冲突，只要镜像层内容不同，UUID 就不一样。同时，在 pull、push、load 和 save 等操作中，加密哈希算法也保证了镜像数据的完整性。

在 Docker 1.10 以前的版本中，不能在镜像之间共享镜像层。采用全新的镜像存储方式以后，在 Docker 1.10 版本中，只要镜像层的数据相同，就可以在不同镜像之间共享。

图 8.3 所示是 Docker 1.10 版本中，容器运行时，镜像层和容器层的存储方式。其中，镜像层采用加密哈希算法生成 UUID，容器层采用随机数方式生成 UUID。

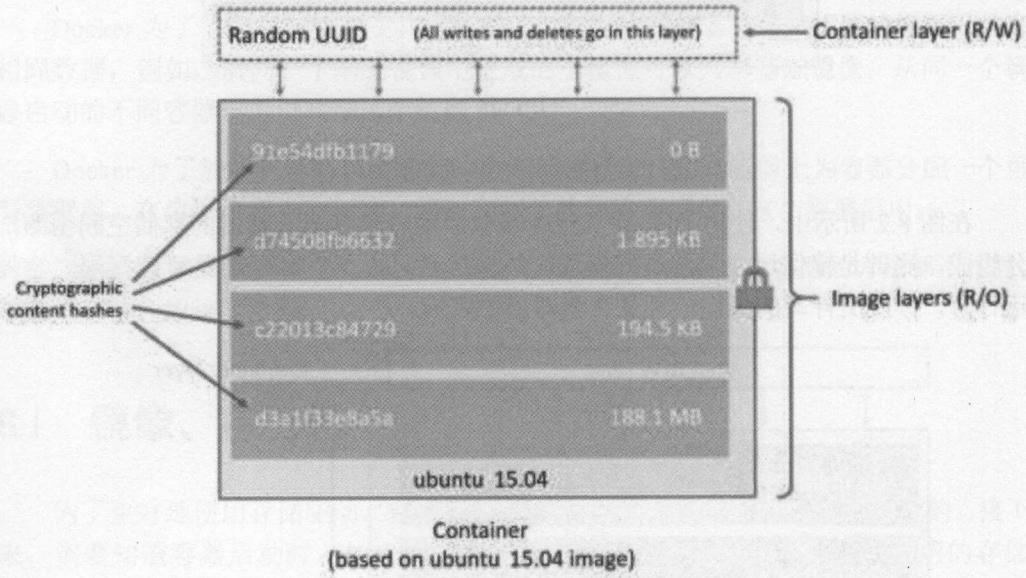


图 8.3　Ubuntu 15.04 的容器结构

采用新的存储方式后，需要注意如下两个方面的变化：

- 把旧版本的镜像迁移到新版本中；
- 镜像层中的文件系统。

当 Docker 升级到 1.10 版本以后，宿主机上会存在一些以前下载的镜像，这些镜像使用旧的存储方式。若要在 Docker 1.10 版本及以后版本使用这些镜像，则需要把这些镜像迁移到新版本中。当 1.10 版本的 Docker Daemon 第一次启动时，会自动迁移镜像。在迁移过程中，Docker 会使用加密哈希算法重新计算每个镜像层的 UUID。迁移完成

以后，每个镜像层都会有新的 UUID。

尽管 Docker Daemon 可以自动迁移镜像，但是在实际应用中，不推荐这种方式。重新计算镜像层的 UUID 会花费很长的时间，特别是当宿主机上有非常多的镜像时。并且，在迁移过程中，Docker Daemon 不能响应其他的 Docker 命令。

可以在升级 Docker 之前，采用 Docker 官方的镜像升级工具完成这项工作。这样，升级 Docker 成功后，Docker Daemon 可以立即响应 Docker 命令。对于集群环境，运维人员也可以并行在多台机器上迁移镜像。

在 Docker Hub 中可以找到这个迁移工具的镜像：

```
# docker pull docker/v1.10-migrator
```

使用这个镜像时，需要把宿主机上存放镜像的目录挂载到容器中。默认情况下，容器镜像放在/var/lib/docker 目录中。

```
# sudo docker run --rm -v /var/lib/docker:/var/lib/docker docker/v1.10-migrator
```

如果在 Docker Daemon 中，则使用 Devicemapper 作为存储驱动，启动迁移工具容器时，需要添加--privileged 选项。

```
# sudo docker run --privileged --rm -v /var/lib/docker:/var/lib/docker docker/
v1.10-migrator
```

8.1.3　一个迁移例子

在这个例子中，Docker 的版本为 1.9.1，使用 AUFS 作为存储驱动。宿主机是 AWS 上 t2.micro 型号的虚拟机，虚拟机配置为 1 个 vCPU，1GB RAM，8GB SSD。所有镜像一共占用了大约 2GB 空间，存放在/var/lib/docker 目录中。

使用 docker images 命令查看当前宿主机上下载的所有镜像。

```
$ docker images
REPOSITORY     TAG            IMAGE ID          CREATED          SIZE
jenkins         latest            285c9f0f9d3d       17 hours ago       708.5 MB
mysql          latest           d39c3fa09ced        8 days ago         360.3 MB
mongo         latest           a74137af4532       13 days ago        317.4 MB
postgres       latest            9aae83d4127f       13 days ago        270.7 MB
redis          latest            8bccd73928d9       2 weeks ago        151.3 MB
centos         latest           c8a648134623        4 weeks ago        196.6 MB
ubuntu         15.04            c8be1ac8145a        7 weeks ago        131.3 MB
```

查看这些镜像一共占用了多少存储空间。

```
$ sudo du -hs /var/lib/docker
2.0G    /var/lib/docker
```

使用迁移容器把上面的 7 个镜像迁移到新版本中，仅用了不到一分钟。其中还包括了从 Docker Hub 上下载 docker/v1.10-migrator 的时间，大约是 3.5s。由此可见，使用工具以后，大大节约了 Docker 升级时间。

```
$ time docker run --rm -v /var/lib/docker:/var/lib/docker docker/v1.10-migrator
Unable to find image 'docker/v1.10-migrator:latest' locally
latest: Pulling from docker/v1.10-migrator
ed1f33c5883d: Pull complete
b3ca410aa2c1: Pull complete
2b9c6ed9099e: Pull complete
dce7e318b173: Pull complete
Digest: sha256:bd2b245d5d22dd94ec4a8417a9b81bb5e90b171031c6e216484db3fe300c2097
Status: Downloaded newer image for docker/v1.10-migrator:latest
time="2016-01-27T12:31:06Z"  level=debug  msg="Assembling  tar  data  for
01e70da302a553ba13485ad020a0d77dbb47575a31c4f48221137bb08f45878d  from  /var/
lib/docker/aufs/diff/01e70da302a553ba13485ad020a0d77dbb47575a31c4f48221137bb0
8f45878d"
time="2016-01-27T12:31:06Z"  level=debug  msg="Assembling  tar  data  for
07ac220aeeef9febf1ac16a9d1a4eff7ef3c8cbf5ed0be6b6f4c35952ed7920d  from  /var/
lib/docker/aufs/diff/07ac220aeeef9febf1ac16a9d1a4eff7ef3c8cbf5ed0be6b6f4c3595
2ed7920d"
time="2016-01-27T12:32:00Z" level=debug msg="layer
dbacfa057b30b1feaf15937c28bd8ca0d6c634fc311ccc35bd8d56d017595d5b took 10.80 seconds"

real    0m59.583s
user    0m0.046s
sys     0m0.008s
```

如果使用性能更好的虚拟机，则可以进一步减少迁移时间。例如，换用 m4.10xlarge 型号的虚拟机，可以把时间控制在 10s 以内。m4.10xlarge 使用 40 个 vCPU，160GB RAM，8GB SSD。

```
real    0m9.871s
user    0m0.094s
sys     0m0.021s
```

8.1.4　容器和容器层

容器和镜像都是由多个层组成，最大的区别在于容器的最上面一层是读写层，叫作容器层。而镜像的所有层都是只读层，叫作镜像层。容器启动后，Docker Daemon 会在容器使用的镜像上添加一个容器层。容器运行时，所有与数据变化相关的操作都是在这个读写层完成的，如新建文件、修改文件等。删除容器时，Docker Daemon 同时会删除这个容器层。必须保留其他的镜像层，因为这些镜像层都是只读的。

每个容器运行时，都有自己的容器层，并在容器层中保存容器运行相关的数据。容器层之下的所有镜像层都是只读的，因此多个容器可以共享同一个镜像。

如图 8.4 所示介绍了从 Ubuntu 15.04 镜像启动多个容器时，镜像层和容器层的使用情况。

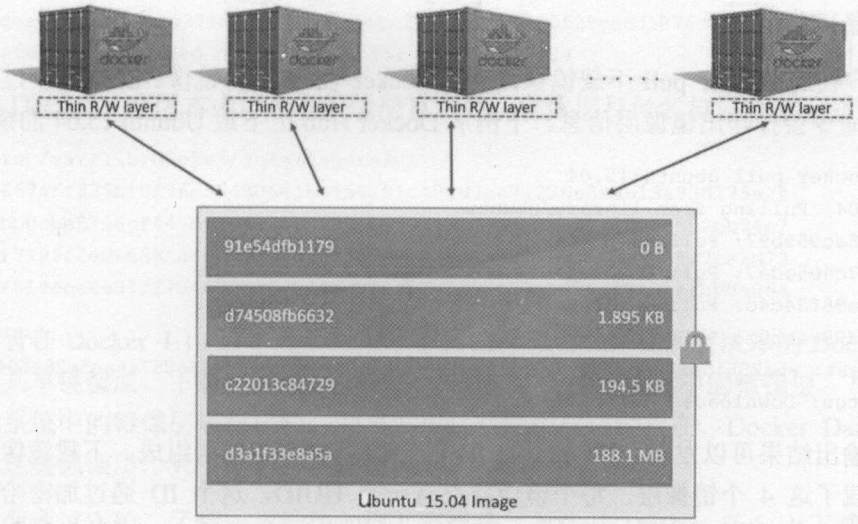


图 8.4　从 Ubuntu 15.04 镜像启动多个容器

Docker 中的存储驱动用于管理镜像层和容器层。不同的存储驱动使用不用的算法和管理方式。在容器和镜像管理中，使用的两大技术是栈式层管理和写时复制（Copy On Write）。

8.1.5　写时复制策略

共享是一种非常流行的资源共享方式，在日常生活中很常见。例如，Jane 和 Joseph 是一对双胞胎，她们对一门代数课程很感兴趣。这门课程是收费项目，为了省钱，她们使用一个人的名义报名缴费。上课时，Jane 和 Joseph 会选择在不同的时候，上不同老师的课。课下，Jane 和 Joseph 会共享一份习题册，她们相互间会复制这份习题册。老师要求 Jane 完成习题册上第 11 页的习题，Jane 会复制第 11 页的习题，完成这些习题，并发送给老师。习题册的内容没有改变，Jane 仅在自己的复制中修改了第 11 页的内容。

写时复制策略也类似，采用了共享和复制。针对相同的数据，系统只保留一份数据，所有操作都访问这一份数据。当有操作需要修改或添加数据时，操作系统会把这部分数据复制到新的地方，这个操作会在新的数据区修改或添加数据。其他操作仍然在旧的数据区读取原始数据。

Docker 在管理镜像和容器时，使用写时复制技术。这项技术节约了镜像的存储空间，加快了容器的启动时间。下面将介绍 Docker 如何通过共享和复制优化镜像和容器。

8.1.6　使用共享技术减小镜像体积

这一节介绍镜像层和写时复制。所有的镜像层和容器层都保存在宿主机的文件系统中，通过 Docker 的存储驱动管理这些层。在 Linux 系统中，宿主机使用/var/lib/docker

保存镜像和容器。

用户使用 docker pull 下载镜像，使用 docker push 上传镜像。在这两个过程中，Docker 命令会打印出镜像层信息。下例从 Docker Hub 上下载 Ubuntu 15.04 的镜像。

```
$ docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

从输出结果可以发现，Ubuntu 15.04 的镜像由 4 个镜像层组成。下载镜像时，实际是下载了这 4 个镜像层。每个镜像层都有一个 UUID，这个 ID 通过加密哈希算法得到。

这些镜像层都保存在宿主机的文件系统中，每层都有独立的目录。在 Docker 1.10 以前的版本中，这些目录的名字与镜像层的 ID 相同。Docker 1.10 及后续版本，采用了新的存储方式。下例是在 Docker 1.9.1 版本中镜像层的存储方式，采用镜像层的 ID 作为目录名字。

在安装了 Docker 1.9.1 版本的主机上下载镜像。

```
$ docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
47984b517ca9: Pull complete
df6e891a3ea9: Pull complete
e65155041eed: Pull complete
c8be1ac8145a: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

在 Docker 1.9.1 版本中，查看镜像层目录名。镜像层目录名为镜像 ID。

```
$ ls /var/lib/docker/aufs/layers
47984b517ca9ca0312aced5c9698753ffa964c2015f2a5f18e5efa9848cf30e2
c8be1ac8145a6e59a55667f573883749ad66eaeef92b4df17e5ea1260e2d7356
df6e891a3ea9cdce2a388a2cf1b1711629557454fd120abd5be6d32329a0e0ac
e65155041eed7ec58dea78d90286048055ca75d41ea893c7246e794389ecf203
```

在 Docker 1.10 及后续版本中，镜像层的目录使用了不同的名字。

在安装了 Docker 1.10 版本的主机上下载镜像。

```
$ docker pull ubuntu:15.04  //下载镜像
15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
```

```
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

在 Docker 1.10 版本中，查看镜像层目录名。镜像层目录名与镜像 ID 不同。

```
$ ls /var/lib/docker/aufs/layers/
1d6674ff835b10f76e354806e16b950f91a191d3b471236609ab13a930275e24
5dbb0cbe0148cf447b9464a358c1587be586058d9a4c9ce079320265e2bb94e7
bef7199f2ed8e86fa4ada1309cfad3089e0542fec8894690529e4c04a7ca2d73
ebf814eccfe98f2704660ca1d844e4348db3b5ccc637eb905d4818fbfb00a06a
```

尽管在 Docker 1.10 版本之后使用了不同的存储方式，在所有版本的 Docker 中，都可以共享镜像层。下载镜像时，Docker Daemon 会检查镜像中的镜像层，与宿主机的文件系统中的镜像层进行比较。如果发现这些镜像层已经存在，Docker Daemon 将会忽略这些镜像层，只下载不存在的镜像层。

下面通过实验，了解一下镜像如何共享数据。首先从 Docker Hub 上下载 Ubuntu 15.04 的镜像。然后用这个镜像作为基础镜像，制作一个新的镜像。

（1）在一个空目录中，新建一个 Dockerfield 文件，使用 Ubuntu 15.04 作为基础镜像。

```
FROM ubuntu:15.04
```

（2）在镜像中，新建一个文件/tmp/newfile，文件中保存“Hello world”字符。

```
FROM ubuntu:15.04
RUN echo "Hello world" > /tmp/newfile
```

（3）保存 Dockerfile。

（4）使用 docker build 命令编译这个镜像，镜像名为 changed-ubuntu。

```
$ docker build -t changed-ubuntu .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:15.04
 ---> 3f7bcee56709
Step 2 : RUN echo "Hello world" > /tmp/newfile
 ---> Running in d14acd6fad4e
 ---> 94e6b7d2c720
Removing intermediate container d14acd6fad4e
Successfully built 94e6b7d2c720
```

新生成的镜像 ID 为 94e6b7d2c720。

（5）新生成的镜像保存在宿主机的文件系统中，使用 docker image 命令查看这个镜像。

```
$ docker images
REPOSITORY        TAG      IMAGE ID       CREATED          SIZE
changed-ubuntu    latest   03b964f68d06   33 seconds ago    131.4 MB
ubuntu            15.04    013f3d01d247   6 weeks ago       131.3 MB
```

（6）使用 docker history 命令查看镜像层的信息。

```
$ docker history changed-ubuntu
IMAGE          CREATED         CREATED BY                                      SIZE                 COMMENT
94e6b7d2c720   2 minutes ago   /bin/sh -c echo "Hello world" > /tmp/newfile     12 B
3f7bcee56709   6 weeks ago     /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>      6 weeks ago     /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/    1.879 kB
<missing>      6 weeks ago     /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   701 B
<missing>      6 weeks ago     /bin/sh -c #(nop) ADD file:8e4943cd86e9b2ca13   131.3 MB
```

从 docker history 的输出可以发现，最上面的镜像层是 94e6b7d2c720。这一层是通过 RUN echo "Hello world" > /tmp/newfile 指令生成的新镜像层。下面的 4 个镜像层是 ubuntu 15.04 镜像中已经存在的层。

在 Docker 1.10 之前的版本中，每个镜像层都有一个配置文件，用于保存镜像层的相关信息。在 Docker 1.10 及后续版本中，使用一个单独的配置文件保存所有镜像层的信息。使用 docker history 命令查看镜像层时，可能会有 missing 的层，这是由于 Docker 版本不同造成的，不用担心。

从图 8.5 所示中可以看出，changed-ubuntu 镜像不会复制下面的 4 个镜像层。这些镜像层在 Ubuntu 15.04 已经存在，changed-ubuntu 和 Ubuntu 15.04 共享这些镜像层。

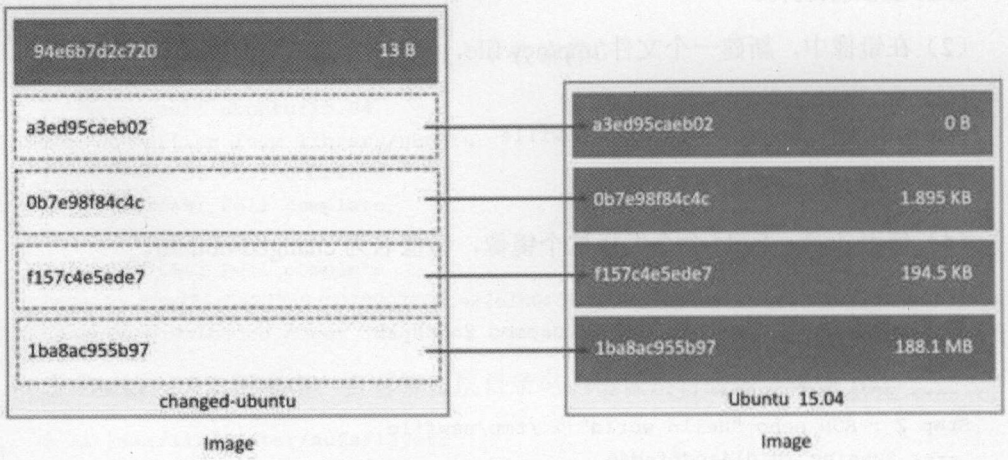


图 8.5　changed-ubuntu 镜像与 Ubuntu 15.04 镜像区别

docker history 命令同时会列出镜像层的大小。changed-ubuntu 中，新建的 94e6b7d2c720 镜像层 12B 的存储空间，这意味着 changed-ubuntu 镜像只会占用额外的 12B 空间，不需要为其他 4 个镜像层分配存储空间。

通过共享技术，Docker 非常节约存储空间。

8.1.7　使用复制技术加快容器启动时间

启动容器时，Docker Daemon 会在镜像层上加一个容器层。图 8.6 所示是从 Ubuntu 15.04 镜像启动容器后，容器层和镜像层的分布。

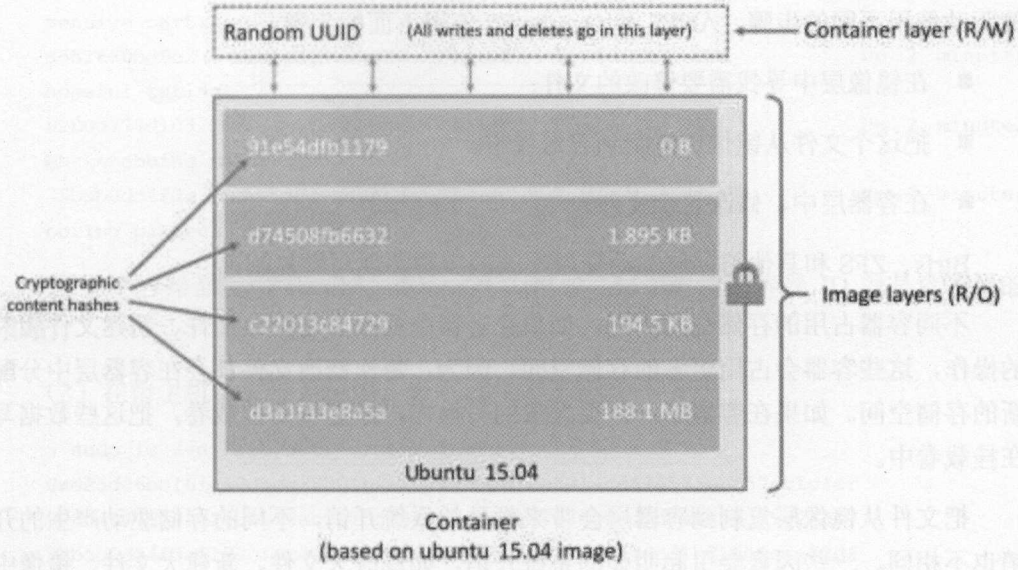


图 8.6　启动容器时，在镜像层上添加一个容器层

在容器中，所有的写操作都发生在容器层中，下面的镜像层都是只读模式，不能修改其中的数据。通过这种方式，多个容器可以共享底部的镜像层。

如图 8.7 所示展示了多个容器共享 Ubuntu 15.04 的镜像层，Ubuntu 15.04 的镜像层在宿主机的文件系统中只保存了一份。每个容器都有自己的容器层，用于保存容器运行的数据。

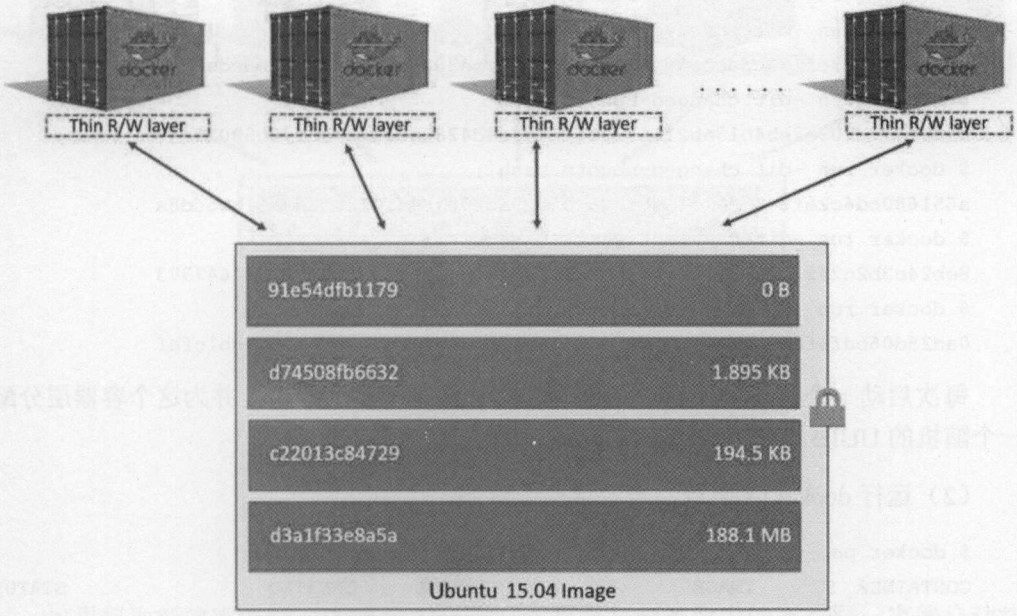


图 8.7　多个容器共享 Ubuntu 15.04 镜像

在容器中修改文件时，Docker 通过存储驱动，发起一个写时复制操作。不同的存

储驱动使用不同的步骤。AUFS 和 OverlayFS 使用下面的步骤：

- 在镜像层中寻找需要修改的文件；
- 把这个文件从镜像层复制到容器层中；
- 在容器层中，修改这个文件。

Btrfs、ZFS 和其他的存储驱动使用不同的步骤实现写时复制。

不同容器占用的存储空间不同。如果在容器中有很多与修改文件、新建文件相关的操作，这些容器会占用更多的存储空间。因为，每次修改文件都会在容器层中分配新的存储空间。如果在容器中，需要大量的写操作，最好使用挂载卷，把这些数据写在挂载卷中。

把文件从镜像层复制到容器层会带来额外的系统开销，不同的存储驱动产生的开销也不相同。一些因素会引起明显的系统开销，如修改大文件、新建大文件、镜像中存在很多镜像层、很深的目录树等。但是，这些系统开销仅发生一次，就是在文件第一次被修改时。一旦 Docker 发现文件被修改，就会把文件从镜像层复制到容器层。其后的操作都是在容器层中完成，不会带来额外开销。

下面通过实验，演示从 changed-ubuntu 镜像启动 5 个容器的过程。

（1）在终端中，使用 docker run 命令启动 5 个容器。这些容器都使用 changed-ubuntu 镜像。

```
$ docker run -dit changed-ubuntu bash
75bab0d54f3cf193cfdc3a86483466363f442fba30859f7dcd1b816b6ede82d4
$ docker run -dit changed-ubuntu bash
9280e777d109e2eb4b13ab211553516124a3d4d4280a0edfc7abf75c59024d47
$ docker run -dit changed-ubuntu bash
a651680bd6c2ef64902e154eeb8a064b85c9abf08ac46f922ad8dfc11bb5cd8a
$ docker run -dit changed-ubuntu bash
8eb24b3b2d246f225b24f2fca39625aaad71689c392a7b552b78baf264647373
$ docker run -dit changed-ubuntu bash
0ad25d06bdf6fca0dedc38301b2aff7478b3e1ce3d1acd676573bba57cb1cfef
```

每次启动一个容器，Docker Daemon 都会添加一个容器层，并为这个容器层分配一个随机的 UUID。这个 UUID 就是 docker run 返回的 ID。

（2）运行 docker ps 命令，查看这 5 个容器。

```
$ docker ps
CONTAINER ID    IMAGE              COMMAND    CREATED              STATUS
PORTS    NAMES
0ad25d06bdf6  changed-ubuntu  "bash"  About a minute ago   Up About a minute
stoic_ptolemy
8eb24b3b2d24  changed-ubuntu  "bash"  About a minute ago   Up About a minute
```

```
pensive_bartik
a651680bd6c2   changed-ubuntu   "bash"   2 minutes ago          Up 2 minutes
hopeful_turing
9280e777d109   changed-ubuntu   "bash"   2 minutes ago          Up 2 minutes
backstabbing_mahavira
75bab0d54f3c   changed-ubuntu   "bash"   2 minutes ago          Up 2 minutes
boring_pasteur
```

这 5 个容器都使用 changed-ubuntu 作为镜像，每个容器的容器 ID 就是容器层的 UUID。

（3）在宿主机的文件系统中，查看这些容器层。

```
$ sudo ls /var/lib/docker/containers
0ad25d06bdf6fca0dedc38301b2aff7478b3e1ce3d1acd676573bba57cb1cfef
9280e777d109e2eb4b13ab211553516124a3d4d4280a0edfc7abf75c59024d47
75bab0d54f3cf193cfdc3a86483466363f442fba30859f7dcd1b816b6ede82d4
a651680bd6c2ef64902e154eeb8a064b85c9abf08ac46f922ad8dfc11bb5cd8a
8eb24b3b2d246f225b24f2fca39625aaad71689c392a7b552b78baf264647373
```

Docker 使用写时复制技术，不仅节约了容器的存储空间，同时也减少了容器的启动时间。启动容器时，Docker Daemon 只需要为每个容器新建一个可写的数据层，而不用复制所有的镜像层。如图 8.8 所示展示了 5 个容器共享一个 changed-ubuntu 镜像。

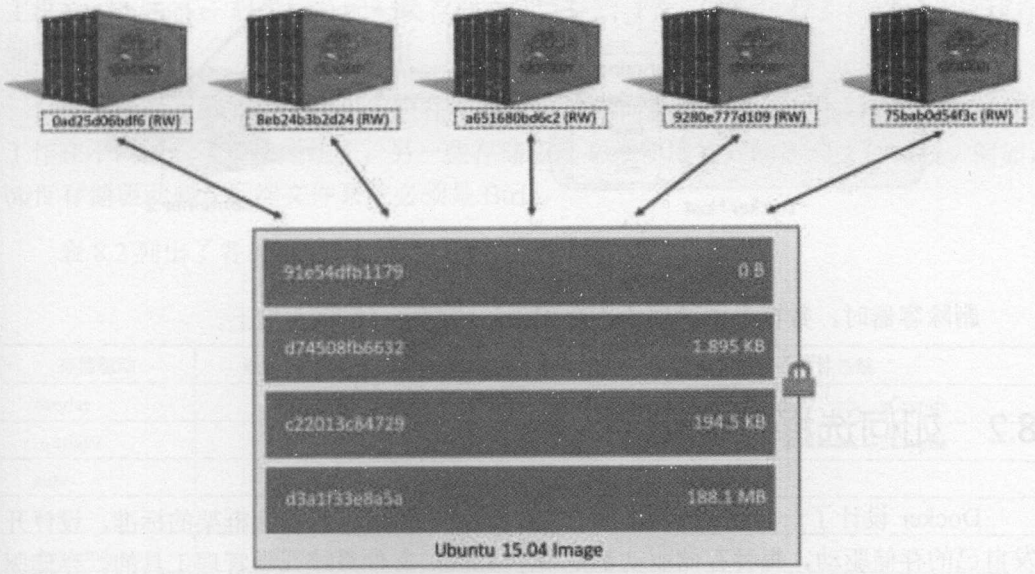


图 8.8　多个容器共享一个镜像

如果每次新建容器时，Docker Daemon 都需要复制镜像中的所有层，容器启动将会非常慢，占用的存储空间也是巨大的。

8.1.8　数据卷和存储驱动

Docker 使用数据卷保证数据持久性。删除容器时，所有不在数据卷中的数据都会被删除。

实际上，数据卷是宿主机上的一个文件或者目录，启动容器时，会把这个文件或目录挂载到容器中。数据卷不受存储驱动程序管理。数据卷中的数据读写操作会绕过存储驱动程序，直接工作在宿主机的文件系统中。容器中挂载的数据卷数量没有限制，多个容器也可以挂载同一个数据卷。

如图 8.9 所示展示了同时启动两个容器，这两个容器共享同一个数据卷。两个容器的容器层在宿主机/var/lib/docker 的不同位置，数据卷都是/data。

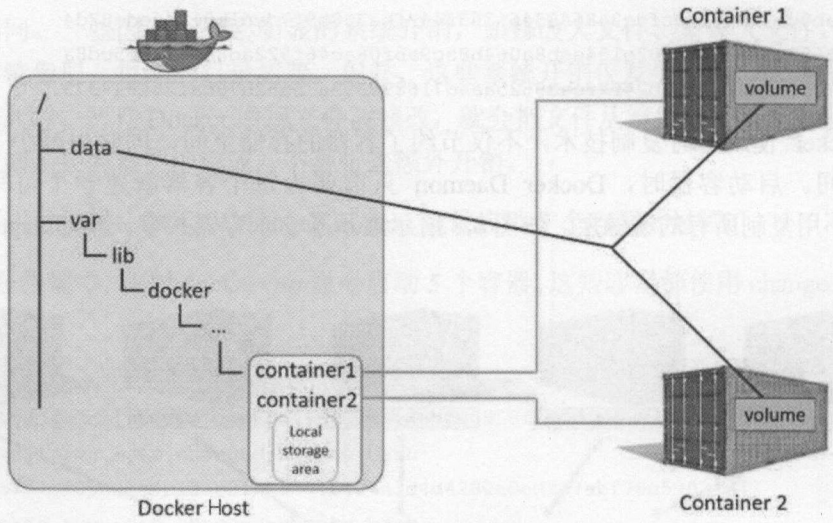


图 8.9　容器共享数据卷

删除容器时，数据卷中的数据不会删除，还是保存在宿主机上。

8.2　如何选择存储驱动

Docker 设计了一套存储驱动插件框架，开发者可以根据这套框架的标准，设计开发自己的存储驱动。每种存储驱动都是基于 Linux 文件系统或卷管理工具的。存储驱动只要实现了框架中的接口，就可以自己管理镜像层和容器层。不同的用户场景需要使用不同的存储驱动，因为不同的存储驱动有自己擅长的方面。

用户需要根据自己的需求选择存储驱动，在 Docker Daemon 中配置特定的存储驱动。该存储驱动会管理所有 Docker Daemon 生成的容器。表 8.1 描述了目前支持的存储驱动类型和存储驱动的名称。

表 8.1　存储驱动类型和存储驱动的名称

存储驱动类型	存储驱动名称
OverlayFS	Overlay 或 Overlay2
AUFS	AUFS
Btrfs	Btrfs
Devicemapper	Devicemapper
VFS	VFS
ZFS	ZFS

通过 docker info 命令可以查看当前环境下使用的存储驱动。

```
$ docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-15-generic
Operating System: Ubuntu 15.04
... output truncated ...
```

在上列中，Docker Daemon 使用 Overlay 作为存储驱动，后端文件系统使用 extfs。后端文件系统表明 Overlay 存储驱动工作在 extfs 文件系统上。Docker Daemon 会在宿主机/var/lib/docker 目录下分配一块存储空间，使用指定的后端文件系统格式化这块存储空间。

使用哪种存储驱动取决于用户在宿主机上使用何种文件系统。一些存储驱动可以工作在不同的后端文件系统上，另一些存储驱动必须使用相同的后端文件系统。例如，btrfs 存储驱动要求后端文件系统必须是 Btrfs。

表 8.2 列出了各种存储驱动和它支持的后端文件系统。

表 8.2　不同存储驱动支持的文件系统

存储驱动	通常使用的后端文件系统	不支持的后端文件系统
overlay	ext4、xfs	btrfs、aufs overlay、overlay2、zfs、eCryptfs
overlay2	ext4、xfs	btrfs、aufs overlay、overlay2、zfs、eCryptfs
aufs	ext4、xfs	btrfs、aufs、eCryptfs
btrfs	btrfs only	N/A
devicemapper	direct-lvm	N/A
vfs	debugging only	N/A
zfs	zfs only	N/A

可以在启动 Docker Daemon 的时候，加入--storage-driver=<name>，设置存储驱动。

下例中，从命令行启动 Docker Daemon，使用 Devicemapper 作为存储驱动。

```
$ docker daemon --storage-driver=devicemapper &
```

查看 Docker Daemon 的运行信息，Storage Driver 是 devicemapper。

```
$ docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-252:0-147544-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.821 GB
 Data Space Total: 107.4 GB
 Data Space Available: 3.174 GB
 Metadata Space Used: 1.479 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.146 GB
 Thin Pool Minimum Free Space: 10.74 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.90 (2014-09-01)
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-15-generic
Operating System: Ubuntu 15.04
```

使用不同的存储驱动可能会影响到容器的性能，因此必须了解不同存储驱动的特点和使用场景。

8.2.1　存储设备和存储驱动

大部分企业都使用专业的 SAN 或者 NAS 作为后端存储，存储阵列可以提供海量存储、高速 IO 及其他高级功能，如按需分配、消重、多点备份、压缩等。

存储驱动和数据卷可以使用存储阵列作为存储设备，此时，Docker 可以使用存储阵列的高级功能。但是，Docker 本身并不能管理这些存储阵列。

Docker 中的每种存储驱动都是基于 Linux 文件系统或者卷管理工具。在使用存储阵列时，要遵循一些现有的最佳实践。

8.2.2　如何存储驱动

选择存储驱动时，需要考虑一些因素。其中，最重要的两点如下：

- 没有哪种存储驱动能够适用于所有场景；
- 每种存储驱动都在不断升级。

可以从以下几方面选择存储驱动。

1．稳定性

使用安装时默认的存储驱动。在不同的 Linux 发行版中安装 Docker 时，安装程序会根据操作系统的不同，选择对应的存储驱动。安装程序自动选择存储驱动时，首先考虑的是稳定性，所有默认的存储驱动是最稳定的。不使用默认的存储驱动则可能会引起故障。

2．熟悉性

选择开发者熟悉的存储驱动，最好是开发者以前用过的存储驱动。如果开发者工作在 Redhat 或 Redhat 的变异版本上，对 LVM 和 Devicemapper 非常熟悉，那么应该选择 Devicemapper 作为存储驱动。

如果开发者对 Docker 中使用的存储驱动都不熟悉，那么就应该选择默认的存储驱动。

3．成熟性

有一些开发者考虑使用 OverlayFS 作为存储驱动，这会带来风险。因为 OverlayFS 不成熟，与 AUFS 和 Devicemapper 相比，也不够稳定，可能会引起更多的故障。开发者在使用 OverlayFS 时，应该加倍小心。

如图 8.10 所示列出了各种存储驱动的优缺点，开发者在选择存储驱动时，应该结合本书前面几节的介绍，综合考虑。

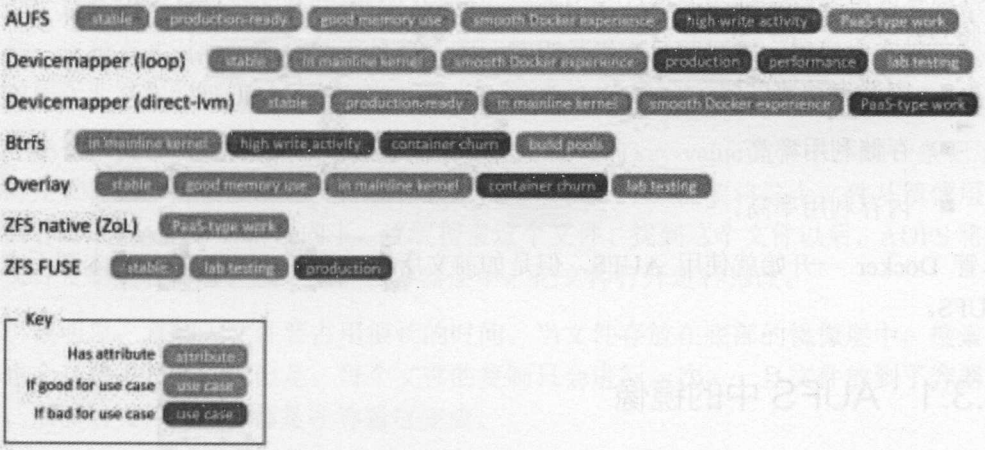


图 8.10　各种存储驱动的优缺点

4．Overlay 和 Overlay2

Overlay 有两种方式，分别是 Overlay 和 Overlay2。这两种方式都使用 OverlayFS 技术，但是在实现方式上不同，并且两种方式也不兼容。因为不兼容，在这两种方之间切换时，之前的镜像不能保存，需要重新下载镜像。Overlay 是原生的实现方式，在 Docker 1.11 及以下版本，是 OverlayFS 的唯一选择。Overlay 方式在 inode 的数量上有

限制，在执行 docker commit 命令时，性能也一般。Overlay2 在 Overlay 的基础上，改进了局限性，提高了性能。但是 Overlay2 仅支持 Linux 内核 4.0 及其后的版本。使用 4.0 之前内核版本或者已经使用了 Overlay 的开发者，建议继续使用 Overlay。使用 4.0 及之后版本内核的开发者，如果不需要以前的镜像，建议可以使用 Overlay2 方式。

8.3　AUFS 存储驱动

AUFS 是一种 Union FS，将不同的目录合并成一个目录，做成一个虚拟文件系统。AUFS 为每个目录设置不同权限，并且可以实时的添加、删除、修改已经挂载好的目录。AUFS 由冈岛顺治郎（Junjiro Okajima）在 2006 年开发，他完全重写了早期的 Union FS，主要目的是增强可靠性、提高性能，并且引入了一些新的功能，如可写分支的负载均衡。AUFS 完全兼容 Union FS，而且比 1.0 版本 Union FS 在稳定性和性能上都要好很多，后来的 UnionFS 2.0 版本开始加入 AUFS 的一些功能。令很多人费解的是 AUFS 没有进入 Linux 主干，原因是 Linus 不允许。Linus 是完美主义者，他认为 AUFS 代码量太大，而且代码质量很烂。Union FS 大概有 10000 行代码，VFS 有 6000 行左右，Union Mount 只有 3000 行，AUFS 居然有 30000 行代码。后来，冈岛不断地改进代码质量，不断地提交，不断地被 Linus 拒绝。最终 AUFS 也没有进入 Linux 主干。不过，目前很多 Linux 发行版都支持 AUFS，如 Ubuntu、Debian 等。

AUFS 是 Docker 最早使用的存储驱动，是目前使用时间最长的一种存储驱动。AUFS 非常稳定，在生产环境中使用广泛，背后有很强大的社区支持。AUFS 有几个显著特点：

- 容器启动速度快；
- 存储利用率高；
- 内存利用率高。

尽管 Docker 一开始就使用 AUFS，但是如前文所述，很多 Linux 发行版本都不支持 AUFS。

8.3.1　AUFS 中的镜像

AUFS 是一个联合文件系统。在 Linux 系统中，AUFS 使用多个目录，叠加起来组成一个单一的文件系统。在实现中，AUFS 使用了联合挂载技术。AUFS 把这些目录通过栈方式组装起来，挂载在一个单一的挂载点，对外提供服务。所有的目录都必须存在于同一台 Linux 主机上。在 AUFS 中，每个目录都叫作一个分支。

在 Docker 中，AUFS 使用联合挂载技术管理镜像层。AUFS 中的每个目录对应了镜像中的一层。

如图 8.11 所示展示了在 AUFS 中，ubuntu:latest 镜像层的结构图。其中，每个镜像

层和容器层都是/var/lib/docker 目录下的一个子目录。联合挂载点对外提供了一个统一的访问路径。在 Docker 1.10 及后续版本中，镜像层的 ID 不再与目录名字一致。

AUFS 同时支持写时复制功能，这项功能在某些存储驱动中是不支持的。

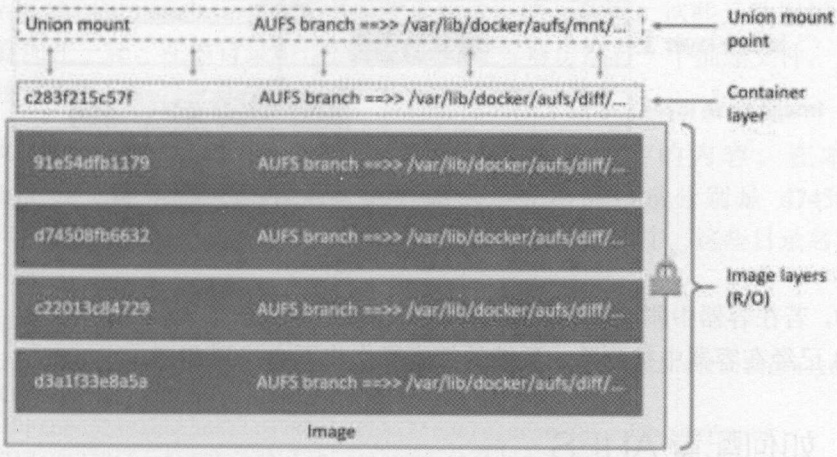


图 8.11　AUFS 中的镜像结构

8.3.2　AUFS 中的容器读写

Docker 使用 AUFS 的写时复制技术实现共享镜像层和节约存储空间。AUFS 工作在文件层，这就意味着即使文件中只有很小的变化，AUFS 的写时复制也会复制整个文件。这种方式对容器的性能有显著影响，特别是当文件特别大、镜像中有很多层或目录层级非常深的时候。

举一个例子，容器中的应用程序需要在一个很大的 key-value 文件中添加一条记录。此时是文件第一次被改动，文件还存放在镜像层。AUFS 需要将这个文件从镜像层复制到容器层。AUFS 会自顶向下，逐层搜索这个文件。找到这个文件以后，AUFS 将整个文件复制到容器层。接下来，在容器层中，把文件打开进行修改。

很明显，复制大文件要占用很长的时间。当文件存放在底部的镜像层中，搜索文件也会花费很长时间。但是，每个文件的复制只会进行一次，一旦文件放到了容器层中，后面对文件的操作都是在容器层完成。

8.3.3　在 AUFS 中删除文件

在 AUFS 中删除文件时，AUFS 会在容器层中生成一个空白文件。该文件会覆盖镜像层中的对应文件，实现假删除。在镜像层中，该文件还是存在，只是在容器中看不到删除的文件。图 8.12 所示简单描述了这个过程。

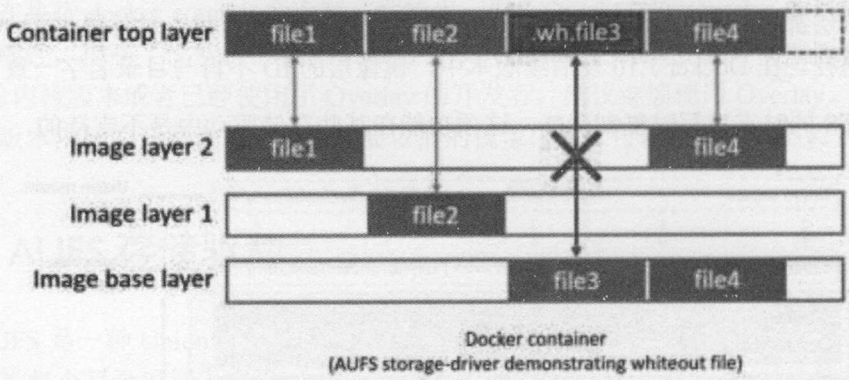


图 8.12　在 AUFS 中删除文件

例如，若在容器中删除 file3，AUFS 会在容器层建立一个.wh.file3 的文件。该文件表示 file3 已经在容器中被删除，后续的搜索操作将不会找到 file3。

8.3.4　如何配置 AUFS

可以在安装了 AUFS 的 Linux 系统中，使用 AUFS 存储驱动。首先，检查系统中是否支持 AUFS。

```
$ grep aufs /proc/filesystems
nodev   aufs
```

接下来，在 Docker Daemon 中配置存储驱动。可以通过命令行启动 Dokcer Daemon。

```
$ sudo docker daemon --storage-driver=aufs &
```

也可以在 Docker Daemon 的配置文件中，配置存储驱动。

```
# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--storage-driver=aufs"
```

使用 docker info 命令，检查 Docker Daemon 使用的存储驱动。

```
$ sudo docker info
Containers: 1
Images: 4
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 6
 Dirperm1 Supported: false
Execution Driver: native-0.2
...output truncated...
```

8.3.5　镜像的存储方式

当 Docker Daemon 使用 AUFS 作为存储驱动时，镜像和容器都保存在宿主机的

/var/lib/docker/aufs 目录下。

镜像层保存在/var/lib/docker/aufs/diff 目录中，Docker 1.10 及后续版本中，镜像层的 UUID 与/var/lib/docker/aufs/diff 中的子目录名不对应。

/var/lib/docker/aufs/layers/目录中保存了镜像层的元数据，这些元数据描述了镜像层如何叠加在一起。在该目录中，每个镜像层和容器层都有一个描述文件，文件中记录了在该层之下所有镜像层的名字。

可以检查/var/lib/docker/aufs/layers/目录中元数据文件的内容。在本例中，91e54dfb11 层之下有三个镜像层，这些镜像层使用的目录分别是 d74508fb66、c22013c847 和 d3a1f33e8a。切记，在 Docker 1.10 及后续版本中，这些目录名与镜像的 UUID 不相同。

```
$ cat
/var/lib/docker/aufs/layers/91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177
d920813e267c
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
```

在镜像层中，最下面一层的元数据文件为空，因为该层下没有其他层。

8.3.6　容器的存储方式

容器运行时，容器中的文件系统被挂载在/var/lib/docker/aufs/mnt/<container-id>。AUFS 把镜像层和容器层叠加起来，通过联合挂载技术，提供一个统一的挂载点。不同容器有不同的挂载点，如/var/lib/docker/aufs/mnt/e9adce3d6577。如果容器没有运行，在宿主机上也会有/var/lib/docker/aufs/mnt/<container-id>，只是该目录为空。这是因为 AUFS 存储驱动只会在容器运行时，才会挂载容器层和镜像层。在 Docker 1.10 及后续版本中，容器 ID 与/var/lib/docker/aufs/mnt/<container-id>中的 container-id 不相同。

下例中，在 Docker 1.10 版本中使用 AUFS 作为存储驱动。

```
$ docker info
Containers: 1
 Running: 1
 Paused: 0
 Stopped: 0
Images: 60
Server Version: 1.10.3
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 82
 Dirperm1 Supported: false
```

启动一个 Ubuntu 14.04 的容器。容器的 ID 为 f420ecd060ba。

```
# docker run -it --rm ubuntu:14.04 bash
root@f420ecd060ba:/#
```

在容器中，新建一个 helloworld.txt 文件。该文件放在容器的根目录中。

```
root@f420ecd060ba:/# touch helloworld.txt
root@f420ecd060ba:/# ls  `pwd`helloworld.txt
/helloworld.txt
```

在宿主机上，进入/var/lib/docker/aufs/mnt/目录。搜索 helloworld.txt 文件，发现该文件存储在/var/lib/docker/aufs/mnt/2e05b3842431 中。AUFS 存储驱动把容器挂载在该挂载点上：

```
# cd /var/lib/docker/aufs/mnt/
# find /var/lib/docker/aufs/mnt -name helloworld.txt
/var/lib/docker/aufs/mnt/2e05b384243110645ed07a44d526a796193926352397b232e57b
74e4b660da89/helloworld.txt
```

从上面的例子可以发现，在 Docker 1.10 版本中，容器 ID 为 f420ecd060ba，挂载目录名为 2e05b3842431，两者不相同。

所有容器在/var/lib/docker/containers/目录中都有自己对应的子目录，子目录名为容器 ID，该目录中包含了运行中的容器和停止的容器。容器的元数据和配置文件保存在/var/lib/docker/containers/<container-id>目录中。当容器运行时，日志文件也保存在<container-id>目录中。

下例中，启动了一个 Ubuntu 14.04 的容器。

```
$ docker ps
CONTAINER ID        IMAGE                                                COMMAND
CREATED             STATUS              PORTS
NAMES
f420ecd060ba        ubuntu:14.04                                         "bash"
19 minutes ago      Up 19 minutes
adoring_archimedes
```

在宿主机的/var/lib/docker/containers/目录中，可以发现容器 f420ecd060ba 的元数据目录。

```
$ ls /var/lib/docker/containers/
f420ecd060ba830dd37a578411664a0b982f9ced69ffd11cdefd0965eaca270f
```

该目录保存了容器的元数据和相关配置文件。其中，f420ecd060ba-json.log 是容器运行时的日志文件。

```
$ ls f420ecd060ba830dd37a578411664a0b982f9ced69ffd11cdefd0965eaca270f/
config.v2.json  hostname   resolv.conf.hash
f420ecd060ba830dd37a578411664a0b982f9ced69ffd11cdefd0965eaca270f-json.log  hosts
shm
hostconfig.json  resolv.conf
```

容器的容器可写层都放在/var/lib/docker/aufs/diff/目录中，每个容器层都有自己对

应的目录。Docker 1.10 及后续版本中，容器 ID 与目录名不一样。容器停止时，容器层目录依然存在。当容器被删除时，容器层目录会被删除。

在前面的例子中，容器 f420ecd060ba 的容器层为 2e05b3842431。在 /var/lib/docker/aufs/diff/目录中，搜索子目录 2e05b3842431。

```
$ find /var/lib/docker/aufs/diff/ -name 2e05b3842431
/var/lib/docker/aufs/diff/2e05b3842431
```

检查该目录内容，发现其中保存了新建的 helloworld.txt 文件。

```
# ls /var/lib/docker/aufs/diff/2e05b3842431
helloworld.txt
```

8.3.7　AUFS 的性能

在 PaaS 的应用场景中，AUFS 是一种很好的选择，因为 AUFS 可以在多个运行中的容器间共享镜像，加速容器启动，节约存储空间。

1. 页缓存

AUFS 在镜像层和容器层中共享文件时，在底层使用了系统的分页缓存技术。这种机制非常高效。

2. 从镜像层复制文件到容器层

AUFS 对容器中写操作的性能有很大的影响，因为 AUFS 存储驱动需要在镜像层中搜索文件，并把文件复制到容器层中。

3. 使用数据卷

数据卷可以提供最好的数据读写性能。使用数据卷后，所有读写操作都会绕过存储驱动，不会增加额外开销。因此，在容器中，当需要大量写操作时，最好把这些数据保存在数据卷中。

8.4　Devicemapper 存储驱动

Devicemapper 是 Linux 2.6 内核提供的一套逻辑卷管理框架，Linux 中的 lvm2、EVMS、dmraid、dm-crypt 都是基于该框架实现的。Docker 中的 Devicemapper 存储驱动也是基于该框架实现的，使用了按需分配和快照功能管理镜像和容器。

最早 Docker 是跑在 Ubuntu 和 Debian 操作系统上，使用 AUFS 作为存储驱动。后来 Docker 流行开来，许多公司希望在 Redhat 上使用 Docker。因为 AUFS 没有进入 Linux 内核，在 Redhat 中不能使用 AUFS。于是 Redhat 的开发者们设计出一套全新的存储驱动，使用 Devicemapper 作为引擎。

Redhat 公司非常看好 Docker，希望 Docker 能够在 Redhat 上跑起来。于是 Redhat 公司和 Docker Inc 公司紧密合作，一起开发适用于 Redhat 的存储驱动。为此，Docker 专门重新设计了 Docker Engine，加入了存储驱动插件框架。最终，两家公司一起开发出 Devicemapper 存储驱动，成为 Docker 中支持的第二种存储驱动。

Linux 内核的维护者在 Linux 内核 2.6.9 中，加入了 Devicemapper 框架，此后的 Redhat 发行版本中都加入了该项功能。Devicemapper 在生产环境中有广泛的应用，同时也得到了 Linux 社区的强力支持，因此 Devicemapper 存储驱动非常稳定。

8.4.1　Devicemapper 中的镜像

Devicemapper 把镜像和容器存储在虚拟设备上，使用按需分配、写时复制快照技术管理镜像和容器。Devicemapper 是对块设备进行操作，而不是整个文件。目前，Cent OS、Redhat 和 Fedora 默认使用 Devicemapper，Ubuntu 和 Debian 默认使用 AUFS，但是也支持 Devicemapper。

Devicemapper 创建镜像的流程如下。

（1）创建一个 thin pool，这个 pool 既可以创建在块设备上，也可以创建在 sparse 文件上。

（2）创建一个基础设备，该设备是一个按需分配的设备，并在该设备上创建文件系统。可以用过 docker info 查看文件系统类型。

```
# docker info
Storage Driver: devicemapper
 Pool Name: gc--docker--vg-gc--docker--thin--pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 21.47 GB
 Backing Filesystem: xfs
```

（3）创建镜像时，会在基础设备上做快照。每创建一个镜像层，就会做一次快照。这些快照使用写时复制技术。快照刚创建完时，不占用存储空间。当内容变化时，才会在 thin pool 中分配存储空间，保存数据。这种技术使创建快照非常迅速。

容器是镜像的一个实例，每个容器都是从一个镜像创建起来的。在 Devicemapper 方式中，容器是镜像的一个快照。容器也使用了按需分配和写时复制技术。当容器中数据变化时，Devicemapper 会从 thin pool 中分配存储空间，保存这些数据。

如图 8.13 所示描述了在 thin pool 上创建基础设备和两个镜像的过程。

从图 8.13 中可以看出，镜像是分层设计的，每个镜像层都是在前一个镜像层上做了一个快照。在镜像中，第一个镜像层是在基础设备上做了一个快照。基础设备是一个 Devicemapper 的设备，而不是一个 Docker 镜像层。

容器是镜像的快照。图 8.14 展示了分别从两个镜像生成不同容器的过程。

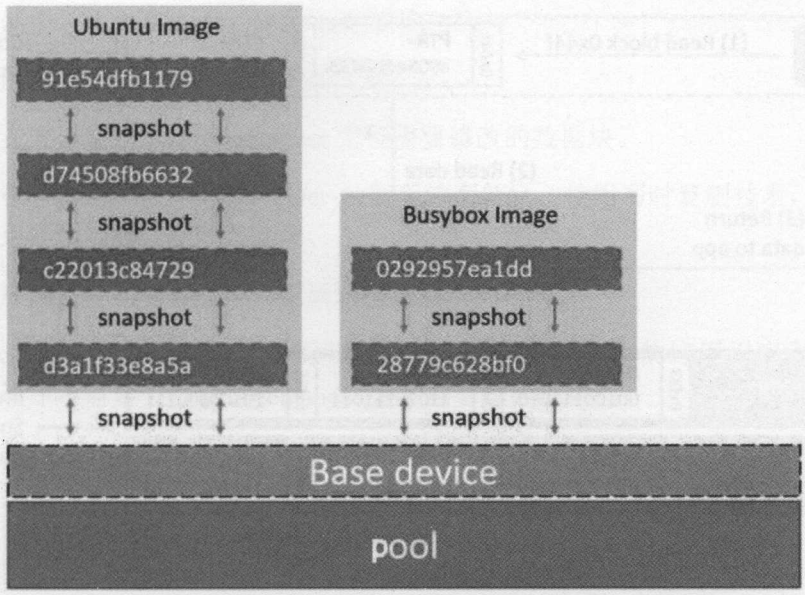


图 8.13　Devicemapper 中的 pool 和镜像层

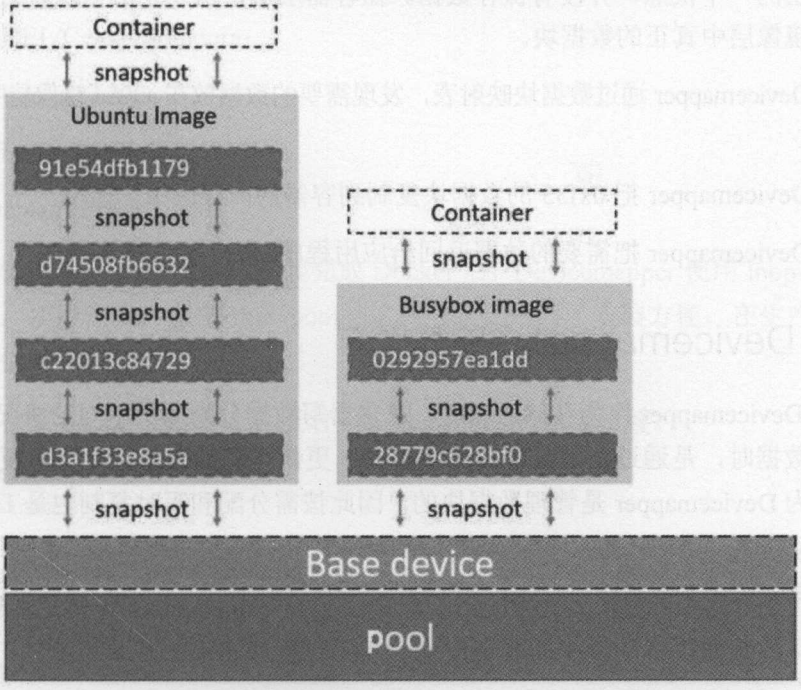


图 8.14　Devicemapper 中的 pool、镜像层和容器层

8.4.2　Devicemapper 中的读操作

本节演示 Devicemapper 中的读操作。图 8.15 展示了在容器中读取一个数据块的流程。

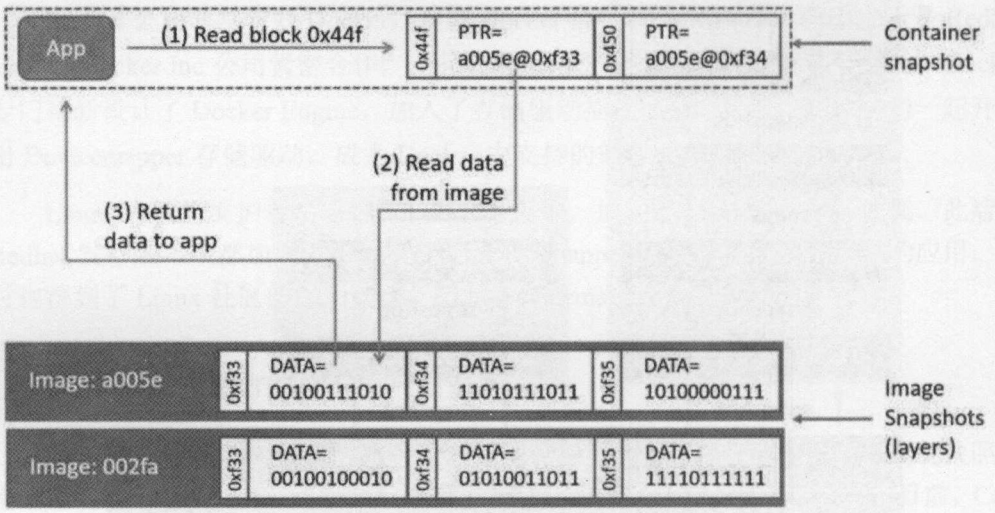


图 8.15　Devicemapper 中的读操作

（1）在容器中，应用程序发起读请求，要求访问地址为 0x44f 的数据块。容器层只是镜像层的一个快照，并没有保存数据。在容器层中，保存了数据块映射表，通过指针指向镜像层中真正的数据块。

（2）Devicemapper 通过数据块映射表，发现需要的数据放在 a005e 镜像层中的 0xf33 数据块上。

（3）Devicemapper 把 0xf33 的数据块复制到容器的内存区中。

（4）Devicemapper 把需要的数据返回给应用程序。

8.4.3　Devicemapper 中的写操作

使用 Devicemapper 作为存储后端时，在容器写数据分为两种不同的情况。当在容器中写新数据时，是通过按需分配技术实现的。更新已有数据是通过写时复制技术实现的。因为 Devicemapper 是管理数据块的，因此按需分配和写时复制也是工作在数据块层。

例如，在容器中，更新大文件的一小部分时，Devicemapper 不需要复制整个文件，仅复制变化的数据块。在 Devicemapper 中，每个数据块的大小是 64KB。

在容器中，写一块 56KB 的新数据，流程如下。

（1）应用程序请求在容器中，写入 56KB 的新数据。

（2）Devicemapper 在容器层中，使用按需分配技术，分配一块 64KB 的数据块。如果请求的数据大于 64KB，Devicemapper 会分配多个数据块。

（3）应用程序把数据写入新分配的数据块中。

在容器中，第一次更新文件的流程如下。

（1）应用程序请求更新容器中的文件。

（2）在容器层中，Devicemapper 定位需要修改的数据块。

（3）在容器层中，Devicemapper 分配新的存储区。使用写时复制技术，把需要修改的数据块复制到新分配的存储区。

（4）应用程序在容器层中，更新这些数据块的内容。

Devicemapper 使用的按需分配和写时复制技术对容器中的应用程序而言是透明的，应用程序不需要了解 Devicemapper 是如何管理数据块的。

因为存在 Devicemapper 存储驱动，对应用程序的读写性能会有一些影响。

8.4.4　如何配置 Devicemapper

在 Linux、Cent OS 等操作系统中，Docker 使用 Devicemapper 作为默认的存储驱动。支持 Devicemapper 的还有如下几个操作系统。

- RHEL/Cent OS/Fedora；
- Ubuntu 12.04；
- Ubuntu 14.04；
- Debian。

默认情况下，在 Redhat 下安装完成 Docker 后，Devicemapper 使用 loop-lvm 模式。该模式下，使用空文件建立 thin pool。这种模式仅是为了安装方便，在生产环境中应该使用其他的模式。

可以使用 docker info 命令，查看 Devicemapper 的详细配置。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-202:2-25220302-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 [...]
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93-RHEL7 (2015-01-28)
 [...]
```

在上例中，Docker 使用 Devicemapper 作为存储驱动，Devicemapper 使用 loop-lvm 模式。从 Data loop file 和 Metadata loop file 的值可以看出，该模式下使用空文件模拟

存储设备。这些文件存储在宿主机的/var/lib/docker/devicemapper/devicemapper 目录下。

8.4.5　在生产环境中配置 direct-lvm 模式

在生产环境中，应该使用 Devicemapper 的 direct-lvm 模式。该模式下，Devicemapper 使用真实的块设备作为存储介质，在块设备上建立 thin pool。

下面将介绍如何使用真实块设备保存镜像和容器，过程如下。

（1）在宿主机上新建一个逻辑卷。

（2）使用逻辑卷作为 Devicemapper 中的 thin pool 的存储介质。

假设在宿主机上有一个闲置的块设备/dev/xvdf，该块设备中有足够的存储空间，如 50GB。在配置 direct-lvm 时，需要停止 Docker Daemon。

（1）登录宿主机，停止 Docker Daemon。

```
$ systemctl stop docker.service
```

（2）安装 lvm2。将使用 lvm2 在 Linux 主机上创建卷组、逻辑卷和其他操作等。

```
$ yum -y update
Loaded plugins: fastestmirror, ovl
base                                                    | 3.6 kB     00:00
extras                                                  | 3.4 kB     00:00
updates                                                 | 3.4 kB     00:00
(1/4): base/7/x86_64/group_gz                           | 155 kB   00:00
(2/4): extras/7/x86_64/primary_db                       | 149 kB   00:00
(3/4): updates/7/x86_64/primary_db                      | 5.7 MB   00:05
(4/4): base/7/x86_64/primary_db                         | 5.3 MB   00:19
Determining fastest mirrors
 * base: mirrors.163.com
 * extras: mirrors.163.com
 * updates: mirrors.zju.edu.cn
Resolving Dependencies
$ yum -y install lvm2
Loaded plugins: fastestmirror, ovl
Loading mirror speeds from cached hostfile
 * base: mirrors.163.com
 * extras: mirrors.163.com
 * updates: mirrors.zju.edu.cn
Resolving Dependencies
--> Running transaction check
---> Package lvm2.x86_64 7:2.02.130-5.el7_2.4 will be installed
--> Processing Dependency: lvm2-libs = 7:2.02.130-5.el7_2.4 for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
--> Processing Dependency: device-mapper-persistent-data >= 0.5.5-1 for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
--> Processing Dependency: liblvm2app.so.2.2(Base)(64bit) for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
```

```
--> Processing Dependency: libdevmapper-event.so.1.02(Base)(64bit) for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
--> Processing Dependency: liblvm2app.so.2.2()(64bit) for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
--> Processing Dependency: libdevmapper-event.so.1.02()(64bit) for package:
7:lvm2-2.02.130-5.el7_2.4.x86_64
--> Running transaction check
……
```

（3）使用/dev/xvdf 创建物理卷。

```
$ pvcreate /dev/xvdf
```

（4）创建卷组，卷组名为 docker。

```
$ vgcreate docker /dev/xvdf
```

（5）创建两个逻辑卷，名为 thinpool 和 thinpoolmeta。其中，thinpool 卷用于存放数据，thinpoolmeta 卷用于存放 thin pool 的元数据。thinpool 卷占用 95%的空间，thinpoolmeta 卷占用 1%的空间，剩余 4%的空间用于扩展卷的空间。

```
$ lvcreate --wipesignatures y -n thinpool docker -l 95%VG
$ lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG
```

（6）使用这两个卷建立一个 thin pool。

```
$ lvconvert -y --zero n -c 512K --thinpool docker/thinpool --poolmetadata
docker/thinpoolmeta
```

（7）在 lvm 的配置文件中，配置自动扩展 thin pool。

```
$ vi /etc/lvm/profile/docker-thinpool.profile
```

（8）设置 thin_pool_autoextend_threshold，单位为%。当逻辑卷中使用的空间超过该值时，lvm 会尝试自动扩展该逻辑卷。把该值设置为 100，表示取消自动扩展功能。

```
thin_pool_autoextend_threshold = 80
```

（9）设置 thin_pool_autoextend_percent，单位为%。该值表示发生自动扩展时，从卷组中划分多少存储空间给改卷。计算方法是：

卷当前的大小×thin_pool_autoextend_percent÷100。

```
thin_pool_autoextend_percent = 20
```

（10）一个完整的/etc/lvm/profile/docker-thinpool.profile 配置如下。

```
activation {
    thin_pool_autoextend_threshold=80
    thin_pool_autoextend_percent=20
}
```

（11）加载/etc/lvm/profile/docker-thinpool.profile 中的配置项。

```
$ lvchange --metadataprofile docker-thinpool docker/thinpool
```

（12）检查系统是否在监控 docker-thinpool 的使用空间。

```
$ lvs -o+seg_monitor
  LV              VG          Attr       LSize  Pool Origin Data%  Meta%  Move
Log Cpy%Sync Convert Monitor
  docker-thinpool docker twi-aotz-- 90.00g                 32.96  44.74
monitored
```

（13）删除/var/lib/docker/目录下的所有文件。这些文件是以前 Docker Daemon 存储的镜像、容器等。因为改变了 Docker Daemon 的存储驱动，所以需要把以前的内容清除干净。

```
$ rm -rf /var/lib/docker/*
```

（14）配置 Docker Daemon，使用 Devicemapper 作为存储驱动。有两种方式，第一种方式是在命令行中启动 Docker Daemon，加入 Devicemapper 相关参数。

```
--storage-driver=devicemapper    --storage-opt=dm.thinpooldev=/dev/mapper/docker-
thinpool --storage-opt dm.use_deferred_removal=true
```

第二种方式是在 Docker Daemon 的配置文件中，添加 Devicemapper 相关参数。

```
{
    "storage-driver": "devicemapper",
    "storage-opts": [
       "dm.thinpooldev=/dev/mapper/docker-thinpool",
       "dm.use_deferred_removal=true"
    ]
}
```

（15）如果采用第二种方式，则需要启动 Docker Daemon。

```
$ systemctl daemon-reload
$ systemctl start docker
```

（16）检查 Docker Daemon 的运行状态。

```
$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor
   preset: disabled)
  Drop-In: /etc/systemd/system/docker.service.d
           └─gcdocker.conf
   Active: active (running) since 一 2016-05-16 09:44:15 CST; 1 months 13 days ago
     Docs: https://docs.docker.com
 Main PID: 2802 (docker)
```

Docker Daemon 启动后，需要定期监控 thin pool 的使用情况。如果 thin pool 的剩余空间不足，则需要在卷组中添加新的块设备。可以使用 lvs 或 lvs -a 监控 thin pool 的剩余空间。

如果发生自动扩展，则 lvm 的扩展过程会记录在系统日志中，可以使用 journalctl 查看系统日志。

```
$ journalctl -fu dm-event.service
```

如果使用 thin pool 时出现问题，则可以使用 Docker Daemon 中的 dm.min_free_space 选项进行调优。

8.4.6　Devicemapper 的存储方式

可以使用 lsblk 命令查看 Devicemapper 创建的 thin pool 及其中的文件。

```
$ sudo lsblk
NAME            MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda            202:0  0    8G  0 disk
└─xvda1             202:1  0    8G  0 part /
xvdf            202:80 0   10G  0 disk
├─vg--docker-data           253:0  0   90G  0 lvm
│ └─docker-202:1-1032-pool 253:2  0   10G  0 dm
└─vg--docker-metadata       253:1  0    4G  0 lvm
  └─docker-202:1-1032-pool 253:2  0   10G  0 dm
```

图 8.16 展示了在 thin pool 中运行两个容器时，容器层和镜像层的层次结构。

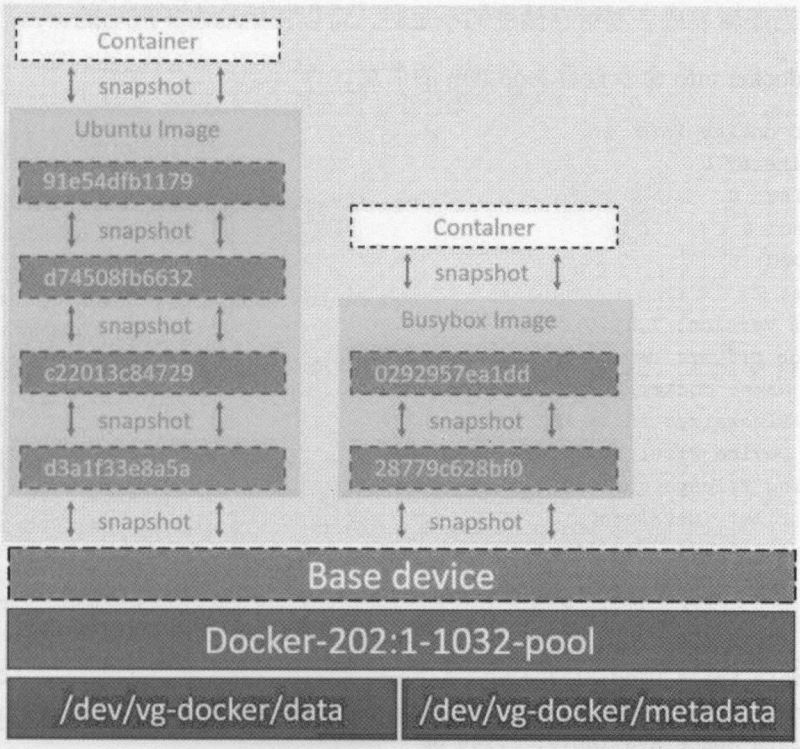


图 8.16　Devicemapper 中的数据卷和元数据卷

图 8.16 中，thin pool 的名字是 Docker-202:1-1032-pool，由两个卷组成，分别是 data 和 metadata。Devicemapper 命名 pool 的格式如下：

```
Docker-MAJ:MIN-INO-pool
```

其中，MAJ 是/挂载点设备的主设备号，上例中是 xvda1 的主设备号。

MIN 是/挂载点设备的次设备号，上例中是 xvda1 的次设备号。

INO 是/var/lib/docker/devicemapper/目录的 inode 号。

```
# ls -id /var/lib/docker/devicemapper/
1032 /var/lib/docker/devicemapper/
```

Devicemapper 是工作在数据块层面，而不是文件层面，在宿主机上很难发现镜像层和容器层的区别。而且，在 Docker 1.10 及后续版本中，镜像层 ID 与/var/lib/docker 中的子目录名字也不相同，找出镜像层对应的文件夹更加困难。不过，有两个子目录需要关注一下，一个是/var/lib/docker/devicemapper/mnt，其中保存了镜像层和容器层的挂载点；另一个是/var/lib/docker/devicemapper/metadata，其中保存了镜像层和容器层的配置文件，配置文件格式为 JSON。

8.4.7　动态扩容 loop-lvm 模式下的 thin pool

在 Docker 运行环境中，可以动态增加 thin pool 的大小，而不需要停止 Docker Daemon。当逻辑卷或者卷组存储空间不足时，可以使用动态扩展功能。

使用 docker info 命令查看 loop-lvm 的信息。

```
$ sudo docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 2
Server Version: 1.11.0-rc2
Storage Driver: devicemapper
 Pool Name: docker-8:1-123141-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: ext4
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.202 GB
 Data Space Total: 107.4 GB
 Data Space Available: 4.506 GB
 Metadata Space Used: 1.729 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.146 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Deferred Deletion Enabled: false
 Deferred Deleted Device Count: 0
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 WARNING: Usage of loopback devices is strongly discouraged for production use.
Either use
```

```
`--storage-opt dm.thinpooldev` or use `--storage-opt dm.no_warn_on_loop_
devices=true` to suppress this warning.
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.90 (2014-09-01)
Logging Driver: json-file
[...]
```

Data Space Total 表明这个 thin pool 的大小为 100GB，下面将把 thin pool 动态扩容到 200GB。

（1）查看 Devicemapper 中设备的存储空间。

```
$ sudo ls -lh /var/lib/docker/devicemapper/devicemapper/
total 1175492
-rw------- 1 root root 100G Mar 30 05:22 data
-rw------- 1 root root 2.0G Mar 31 11:17 metadata
```

（2）增加 data 文件到 200GB。

```
$ sudo truncate -s 214748364800 /var/lib/docker/devicemapper/devicemapper/data
```

（3）检查文件大小变化。

```
$ sudo ls -lh /var/lib/docker/devicemapper/devicemapper/
total 1.2G
-rw------- 1 root root 200G Apr 14 08:47 data
-rw------- 1 root root 2.0G Apr 19 13:27 metadata
```

（4）重新加载 loop 设备。

```
$ sudo blockdev --getsize64 /dev/loop0
107374182400
$ sudo losetup -c /dev/loop0
$ sudo blockdev --getsize64 /dev/loop0
214748364800
```

（5）获取 thin pool 的名字。名字为“:之前的所有字符”，本例中为 docker-8:1-123141-pool。

```
$ sudo dmsetup status | grep pool
docker-8:1-123141-pool: 0 209715200 thin-pool 91
422/524288 18338/1638400 - rw discard_passdown queue_if_no_space -
```

（6）打印设备映射表。

```
$ sudo dmsetup table docker-8:1-123141-pool
0 209715200 thin-pool 7:1 7:0 128 32768 1 skip_block_zeroing
```

（7）获得当前 thin pool 使用的扇区数。从 dmsetup table 命令可以看出，当前 thin pool 使用的扇区数为 209715200。当把 thin pool 扩展到 200GB 时，需要把扇区数增加到 419430400。

（8）使用新的扇区数，重新加载 thin pool。

```
$ sudo dmsetup suspend docker-8:1-123141-pool \
   && sudo dmsetup reload docker-8:1-123141-pool --table '0 419430400 thin-pool
```

```
    7:1 7:0 128 32768 1 skip_block_zeroing' \
&& sudo dmsetup resume docker-8:1-123141-pool
```

在 Docker 的源码中，有一个 contrib 目录，其中包含了一些 Docker 相关的工具。这些工具非常有用，但是可能没有更新。其中，有一个工具是 device_tool.go，可以使用这个工具动态扩展 loop-lvm 方式下的 thin pool。

首先在 Golang 语言环境下，编译这个源文件。然后在宿主机上使用命令行动态扩展 thin pool。

```
$ ./device_tool resize 200GB
```

8.4.8　动态扩容 direct-lvm 模式下的 thin pool

（1）扩容卷组 vg-docker。

```
$ sudo vgextend vg-docker /dev/sdh1
Volume group "vg-docker" successfully extended
```

（2）扩容逻辑卷 vg-docker/data。

```
$ sudo lvextend  -l+100%FREE -n vg-docker/data
Extending logical volume data to 200 GiB
Logical volume data successfully resized
```

（3）获得 thin pool 名字。

```
$ sudo dmsetup status | grep pool
docker-253:17-1835016-pool: 0 96460800 thin-pool 51593 6270/1048576 701943/
753600 - rw no_discard_passdown queue_if_no_space
```

（4）打印设备映射表。

```
$ sudo dmsetup table docker-253:17-1835016-pool
0 96460800 thin-pool 252:0 252:1 128 32768 1 skip_block_zeroing
```

（5）使用 blockdev 命令，获得当前 thin pool 使用的扇区数。在本例中需要把卷扩容到 264132100096byte，新扇区数为 515883008。

```
$ sudo blockdev --getsize64 /dev/vg-docker/data
264132100096
```

（6）使用新的扇区数，重新加载 thin pool。

```
$ sudo dmsetup suspend docker-253:17-1835016-pool \
   && sudo dmsetup reload docker-253:17-1835016-pool --table  '0 515883008
thin-pool 252:0 252:1 128 32768 1 skip_block_zeroing' \
&& sudo dmsetup resume docker-253:17-1835016-pool
```

8.4.9　Devicemapper 的性能

按需分配会影响容器的性能。Devicemapper 使用按需分配技术，为容器分配新的数据区。当应用程序需要在容器中写入新数据时，Devicemapper 会从 thin pool 中划分

新的数据区，并分配给容器。

Devicemapper 使用的数据块大小为 64KB，每次分配的最小数据为 64KB。即使应用程序请求的数据小于 64KB，Devicemapper 也会分配 64KB 的空间。当应用程序请求的数据大于 64KB 时，Devicemapper 会按照 64KB 的整数倍分配存储空间。分配数据块会影响容器的性能，特别是容器中发生了大量小数据写操作的时候。一旦数据块分配给容器后，后续的读写操作都工作在容器层中。

写时复制也会影响容器的性能。在容器中第一次更新数据时，Devicemapper 会发起写时复制操作，把数据块从镜像层复制到容器层。写时复制对容器性能有明显的影响。

写时复制也使用 64KB 作为一个数据块。当需要在大文件中更新很小的数据时，Devicemapper 比 AUFS 在性能上有显著提高。在 1GB 的文件中，更新 32KB 大小的数据时，Devicemapper 只会复制一个 64KB 的数据块到容器层。而 AUFS 需要复制整个文件。当在容器中，执行大量小数据写操作时，Devicemapper 的性能不如 AUFS，如反复地写小于 64KB 的数据块。

影响 Devicemapper 性能的主要因素有如下几个。

（1）loop-lvm 模式或 direct-lvm 模式。

Devicemapper 默认使用 loop-lvm 模式，该模式使用空文件模拟存储设备。不推荐在生产环境使用这种该模式，因为它的性能非常低。在生产环境中，应该使用 direct-lvm 模式，该模式直接在存储设备上进行读写。

（2）高速存储设备。

为了提高性能，最好使用高速存储设备建立 thin pool，如 SSD。把宿主机连上专业的 SAN 存储阵列或 NAS 存储阵列可以实现这一功能。

（3）内存使用效率。

在 Docker 的存储驱动中，Devicemapper 不是内存利用率最高的方式。从一个镜像启动 *N* 个容器时，会在内存中加载 *N* 份复制的镜像，这会占用宿主机上的内存。在 PaaS 应用场景中，Devicemapper 不是最佳选择。

（4）使用数据卷。

数据卷可以提供最好的数据读写性能。使用数据卷后，所有读写操作都会绕过存储驱动，不会增加额外开销。因此，在容器中，当需要大量写操作时，最好把这些数据保存在数据卷中。

8.5　Btrfs 存储驱动

Btrfs 是下一代使用写时复制技术的文件系统，使用了很多高级存储技术，非常适合在 Docker 中使用。Btrfs 已经进入 Linux 内核主干中，使用的 on-disk-format 技术被

公认为很稳定。但是，很多功能还处于开发阶段，开发者可以考虑等版本稳定以后，把 Docker 迁移上去。

在 Docker 中，Btrfs 存储驱动使用了按需分配、写时复制和快照技术管理镜像和容器。当 Docker 出现在 Linux 上时，很多人发现 Btrfs 是一种潜在的 Devicemapper 替代品。但是，到目前为止，Devicemapper 还是比 Btrfs 更安全、更稳定，也更适合部署在生产环境中。如果开发者有使用 Btrfs 的经验，可以考虑在生产环境部署 Btrfs 作为存储驱动。

8.5.1　Btrfs 中的镜像

Docker 使用 Btrfs 中的子卷和快照技术管理镜像和容器。Btrfs 中的子卷与 UNIX 文件系统类似，每个子卷都有自己的目录结构，可以和 UNIX 文件系统结合起来。

子卷本身使用写时复制技术。分配存储时，采用按需分配技术。子卷可以是一个目录，也可以在其中保存文件。如图 8.17 所示，有 4 个子卷，subvolume2 和 subvolume3 是目录，subvolume4 中保存了目录和文件。

Btrfs 中的快照会复制整个子卷，并且快照是可读写的。快照保存在子卷中，也可以对快照递归的做快照，如图 8.18 所示。

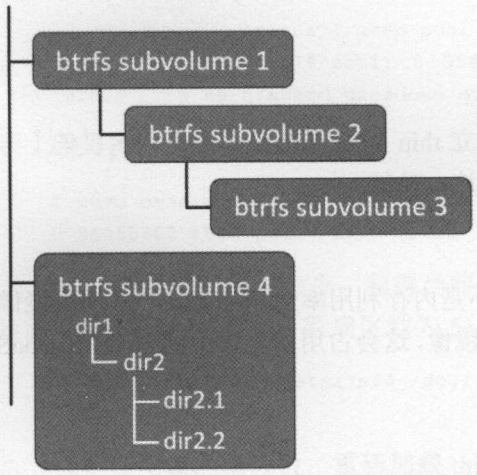


图 8.17　Btrfs 中的目录和文件

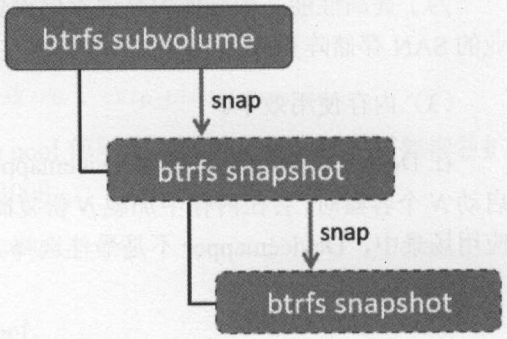


图 8.18　Btrfs 中的快照

Btrfs 使用写时复制技术为子卷和快照分配底层存储空间。分配存储空间时，以块为单位，块大小一般为 1GB。

快照是子卷的第一层，表面上看起来，快照中的操作与子卷中的操作是一样的。Btrfs 在内部提供这种机制，对外透明。Btrfs 在存储空间使用上很高效，对性能也没有影响。图 8.19 展示了子卷和快照如何共享数据。

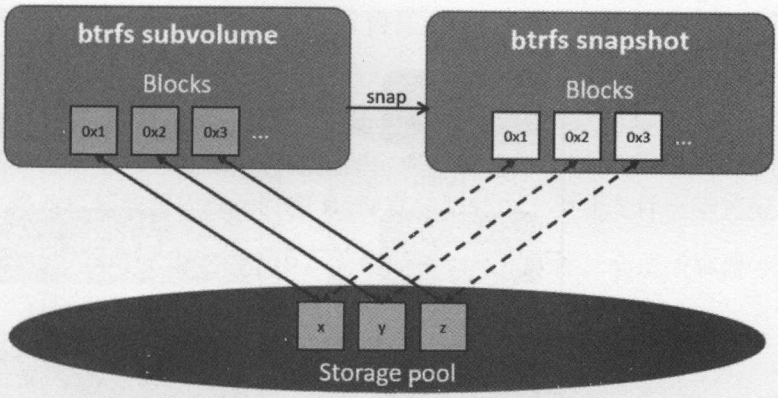


图 8.19　Btrfs 中的数据块

Btrfs 把镜像层和容器层保存在独立的子卷或快照中。镜像中的基础层作为一个子卷保存，其他镜像卷和容器卷都作为快照保存。图 8.20 展示了这种关系。

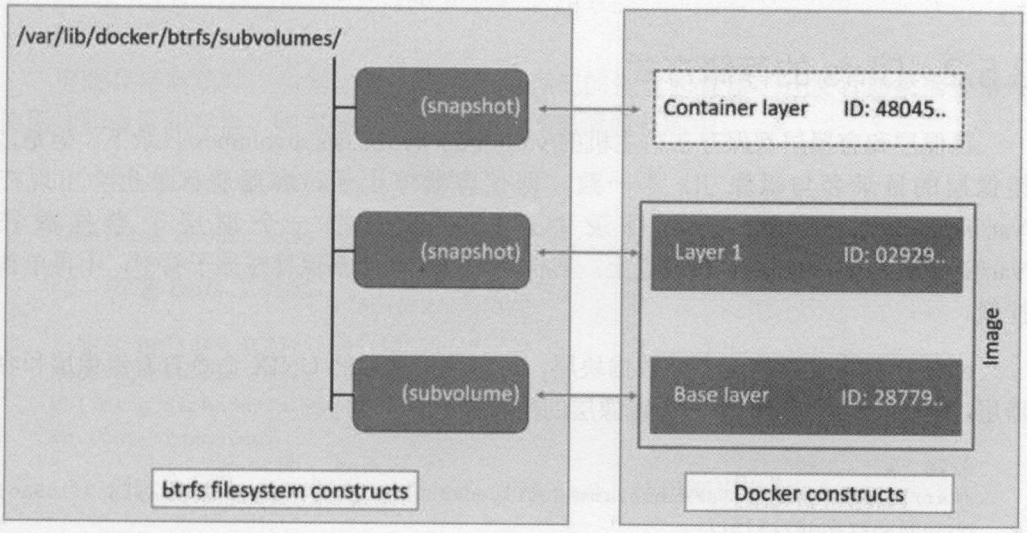


图 8.20　Btrfs 中的镜像层和容器层

Btrfs 创建镜像层和容器层的流程如下。

（1）为基础镜像层创建一个 Btrfs 子卷，保存在/var/lib/docker/btrfs/subvolumes 目录下。

（2）Btrfs 为剩下的镜像层创建一个快照，这个快照建立在上层的镜像层基础之上。图 8.21 展示了一个 3 层镜像。基础层是一个 Btrfs 子卷。Layer1 在基础层之上，是子卷的快照。Layer2 在 Layer1 之上，是快照的快照。

在 Docker 1.10 及后续版本中，镜像 ID 与/var/lib/docker/目录中的子目录名不一致。

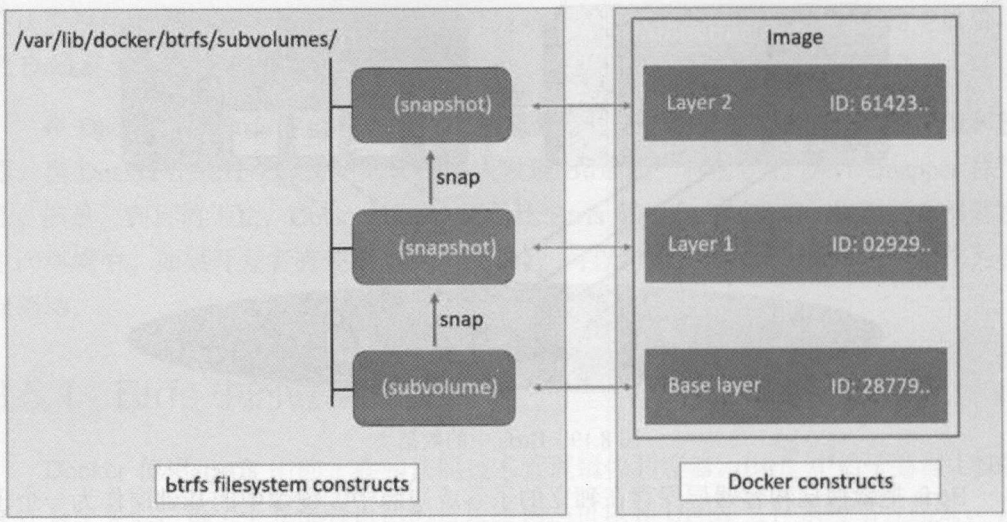


图 8.21　Btrfs 中镜像层的关系

8.5.2　Btrfs 的存储方式

镜像层和容器层都保存在宿主机的/var/lib/docker/btrfs/subvolumes/目录下。但是，镜像层的目录名与镜像 ID 不一致。即使容器停止了，容器层目录也会出现在/var/lib/docker/btrfs/subvolumes/目录中。btrfs 默认把一个顶层子卷挂载在/var/lib/docker/subvolumes 挂载点上。其他的子卷和快照都保存在该子卷中，不再单独挂载。

btrfs 工作在文件层而不是数据块层，可以使用普通的 UNIX 命令查看镜像层和容器层。例如，一个 Btrfs 中一个镜像层的内容如下。

```
$ ls -l
/var/lib/docker/btrfs/subvolumes/0a17decee4139b0de68478f149cc16346f5e711c5ae3
bb969895f22dd6723751/
total 0
drwxr-xr-x 1 root root 1372 Oct  9 08:39 bin
drwxr-xr-x 1 root root    0 Apr 10  2014 boot
drwxr-xr-x 1 root root  882 Oct  9 08:38 dev
drwxr-xr-x 1 root root 2040 Oct 12 17:27 etc
drwxr-xr-x 1 root root    0 Apr 10  2014 home
...output truncated...
```

8.5.3　Btrfs 中的读写

在 Btrfs 中，容器层是镜像层的快照。快照中保存元数据，元数据保存了数据在存储设备中的地址。子卷也使用元数据保存数据在存储设备上的地址。读操作在快照和子卷中的性能没有差别。在 Btrfs 存储驱动中，读文件时不会牺牲性能。

在容器中写新文件时，Btrfs 会使用按需分配技术，为容器层分配新的存储空间。

新文件数据保存在新分配的空间中。Btrfs 内部支持按需分配，因此在 Btrfs 存储驱动中，写新文件时不会牺牲性能。

在容器中更新文件时，Btrfs 存储驱动会发起一个写时复制操作。Btrfs 不会修改原文件，而是在容器层中分配新的存储空间，保存修改的数据。接下来，Btrfs 会更新快照元数据，修改更新数据的地址。在 Btrfs 存储驱动中，更新文件会牺牲很小的性能。

总结一下，在 Btrfs 存储驱动中，大量写和更新小文件，会造成容器性能下降。

8.5.4　如何配置 Btrfs

使用 Btrfs 存储驱动时，Docker 把 Btrfs 文件系统挂载到/var/lib/docker 上。下面介绍在 Ubuntu 14.04 中配置 Btrfs 的过程。

首先检查系统是否支持 Btrfs。

```
$ cat /proc/filesystems | grep btrfs
btrfs
```

更改存储驱动时，会删除宿主机上已经存在的镜像和容器。开发者如果需要继续使用这些镜像和容器，则需要提前把它们上传到镜像仓库中。

（1）停止 Docker Daemon。

```
$ service docker stop
```

（2）安装 Btrfs 工具包。

```
$ sudo apt-get install btrfs-tools
Reading package lists... Done
Building dependency tree
<output truncated>
```

（3）创建 Btrfs pool。使用 mkfs.btrfs 创建 pool，使用裸设备/dev/xvdb。

```
$ sudo mkfs.btrfs -f /dev/xvdb
WARNING! - Btrfs v3.12 IS EXPERIMENTAL
WARNING! - see http://btrfs.wiki.kernel.org before using

Turning ON incompat feature 'extref': increased hardlink limit per file to 65536
fs created label (null) on /dev/xvdb
    nodesize 16384 leafsize 16384 sectorsize 4096 size 4.00GiB
Btrfs v3.12
```

Btrfs 目前还处于开发阶段，在生产环境中要慎重使用。

（4）创建挂载点/var/lib/docker。

```
$ sudo mkdir /var/lib/docker
```

（5）获得 Btrfs 文件系统的 UUID。

```
$ sudo blkid /dev/xvdb
```

```
/dev/xvdb: UUID="a0ed851e-158b-4120-8416-c9b072c8cf47"
UUID_SUB="c3927a64-4454-4eef-95c2-a7d44ac0cf27" TYPE="btrfs"
```

（6）配置系统启动时，自动挂载 Btrfs 文件系统。

在/etc/fstab 中，把 Btrfs 挂载到/var/lib/docker 挂载点上。

```
$ sudo mkdir -p /var/lib/docker
$ sudo vi /etc/fstab
/dev/xvdb /var/lib/docker btrfs defaults 0 0
UUID="a0ed851e-158b-4120-8416-c9b072c8cf47" /var/lib/docker btrfs defaults 0 0
```

（7）挂载 Btrfs 文件系统。

```
$ sudo mount -a
$ mount
/dev/xvda1 on / type ext4 (rw,discard)
<output truncated>
/dev/xvdb on /var/lib/docker type btrfs (rw)
```

完成创建 Btrfs 后，在 Docker Daemon 中配置使用 Btrfs 存储驱动。

（8）在 Docker Daemon 中配置存储驱动。可以通过命令行启动 Dokcer Daemon。

```
$ sudo docker daemon --storage-driver=btrfs&
```

也可以在 Docker Daemon 的配置文件中，配置存储驱动。

```
# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--storage-driver=btrfs"
```

（9）使用 docker info 命令检查 Btrfs 是否生效。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: btrfs
[...]
```

8.5.5　Btrfs 的性能

使用 Btrfs 存储驱动以后，有一些因素会影响容器性能。

1．页缓存

Btrfs 不支持共享页缓存。这会造成 *N* 个容器访问同一文件时，需要在内存中保存该文件的 *N* 个复制文件。在 PaaS 的应用场景下，Btrfs 不是最好的选择。

2．写小文件

在容器中，大量写小文件时，不能充分使用 Btrfs 的块性能，因为 Btrfs 中，块的大小为 1GB 左右。大量写小文件的操作可能会引起宿主机存储空间不足，造成 Docker Daemon 停止。这也是 Btrfs 存储驱动目前最大的缺陷。使用 Btrfs 存储驱动时，运维人员需要定时监控 Btrfs 文件系统中的可用空间，可以使用 btrfs filesys show 命令。一定

要使用 Btrfs 自带的文件系统监控工具，不要使用 UNIX 系统自带的 df 命令，该命令对 Btrfs 文件系统的使用情况统计不准确。

3．顺序写操作

Btrfs 使用日志记录技术写数据。这会影响顺序写操作，会损失约一半的性能。

4．碎片

碎片是写时复制文件系统的一个副产品。大量写数据的随机写操作会带来碎片。表现为在使用 SSD 存储设备时 CPU 出现峰值，使用传统硬盘时出现硬盘振荡，这些都会损失性能。当前版本的 Btrfs 文件系统允许开发者在挂载时，使用 autodefrag 参数。此时，Btrfs 会监控随机写操作，并消除碎片。开发者可以在测试环境中，使用 autodefrag 进行测试。一些测试表明，autodefrag 可以解决碎片问题，但是对在容器进行大量小文件的写操作有影响。

5．SSD

Btrfs 内部对 SSD 设备进行了优化。在挂载 Btrfs 文件系统时，可使用-o ssd 开启该功能，这个参数可以提高 SSD 上的写操作性能。Btrfs 同时也支持禁止 TRIM 功能。可以在挂载 Btrfs 文件系统时，使用-o discard 参数，该选项可能影响性能。建议在测试环境中，使用不同选项进行性能测试。

6．使用数据卷

数据卷可以提供最好的数据读写性能。使用数据卷后，所有读写操作都会绕过存储驱动，不会增加额外开销。因此，在容器中，当需要大量写操作时，最好把这些数据保存在数据卷中。

8.6　ZFS 存储驱动

ZFS 是下一代文件系统，提供卷管理、快照、校验、压缩、消重和多地复制等功能。ZFS 最早是由 Sun 公司开发的，现在已经开源，使用 CDDL 开源协议。因为 CDDL 和 GPL 协议不兼容，ZFS 没有进入 Linux 内核主干。在 Linux 上，可以使用 ZFS on Linux 项目，该项目提供了 ZFS 的管理工具。

如果开发者没有使用过 ZFS，建议不要在生产环境中使用 ZFS 存储驱动。

8.6.1　ZFS 中的镜像

在 Docker 中，使用了三种 ZFS DataSet 扩展，包括文件系统、快照和克隆。

ZFS 使用按需分配技术，从 ZFS pool 中分配存储空间。快照和克隆可以节约 ZFS 的存储空间。其中，快照是只读的，克隆是可读写的。克隆只能从快照上产生。图 8.22 展示了 ZFS 文件系统中快照和克隆的关系。

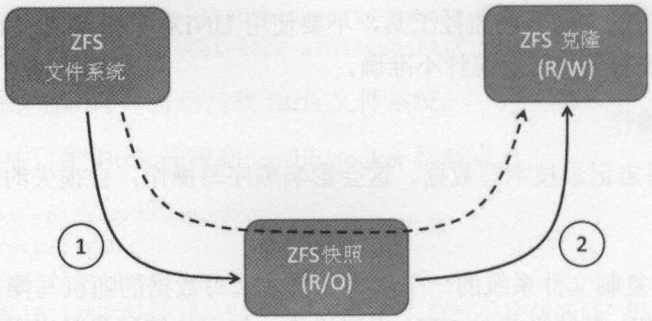


图 8.22　ZFS 文件系统中快照和克隆的关系

实线展示了创建克隆的过程。在步骤①中，系统在文件系统上新建一个快照。在步骤②中，系统从快照创建一个克隆。虚线表示克隆和文件系统的关系。三个 ZFS DataSet 都是从底层的 zpool 中分配存储空间。

在 ZFS 存储驱动中，镜像的基础层是一个 ZFS 文件系统。其他的镜像层是一个 ZFS 克隆，这个克隆是从下面一层的 ZFS 快照得到的。图 8.23 展示了一个两层镜像的镜像层结构图。

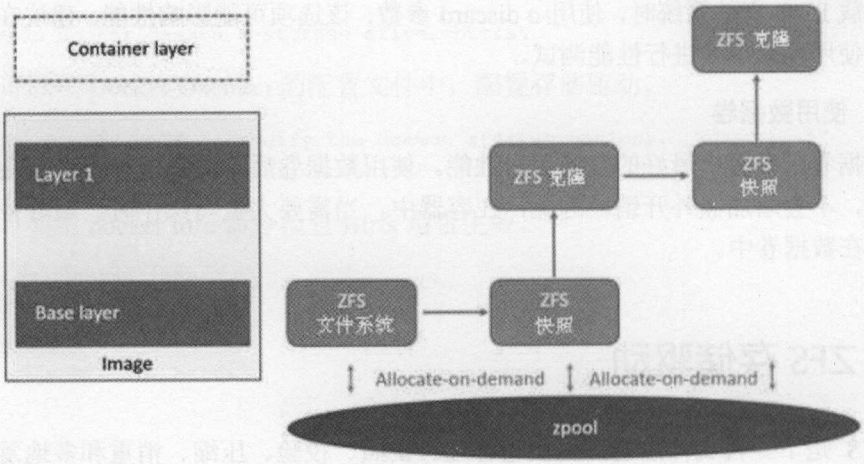


图 8.23　ZFS 中镜像层的关系

下面简单介绍 ZFS 创建镜像层和容器层的过程。

（1）镜像的基础镜像层保存在 ZFS 文件系统上，保存在/var/lib/docker 目录中。

（2）新的镜像层是它下层 DataSet 的一个克隆。

（3）启动容器后，会在顶部增加一个读写层。

8.6.2　ZFS 中的读写

使用 ZFS 存储驱动时容器中的读操作很简单，速度也很快，如图 8.24 所示。

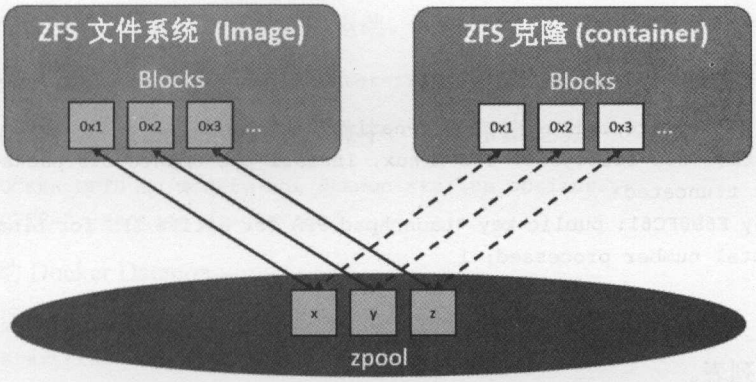


图 8.24　ZFS 中的数据块

在容器中，写新文件是通过按需分配实现的。每次在容器中需要写新文件时，ZFS 存储驱动会从 zpool 中分配新的空间。

在容器中，更新文件时，ZFS 驱动会在容器层中分配新的数据块，保存修改的数据，不会修改镜像层中的数据。

8.6.3　如何配置 ZFS

在 Docker 中使用 ZFS 存储驱动，需要把 ZFS 文件系统挂载到/var/lib/docker 上。下面介绍如何在 Ubuntu 16.04 和 Ubuntu 14.04 中使用 ZFS 存储驱动。

（1）停止 Docker Daemon。

```
$ service docker stop
```

（2）在 Ubuntu 16.04 中安装 ZFS 包。

```
$ sudo apt-get install -y zfs
Reading package lists... Done
Building dependency tree
<output truncated>
```

检查系统是否加载 ZFS 模块。

```
$ lsmod | grep zfs
zfs                  2813952  3
zunicode              331776  1 zfs
zcommon                57344  1 zfs
znvpair                90112  2 zfs,zcommon
spl                   102400  3 zfs,zcommon,znvpair
zavl                   16384  1 zfs
```

（3）在 Ubuntu 14.04 中安装 ZFS 包时，需要使用 software-properties-common 包。

```
$ sudo apt-get install -y software-properties-common
Reading package lists... Done
Building dependency tree
```

```
<output truncated>
```

安装 zfs-native。

```
$ sudo add-apt-repository ppa:zfs-native/stable
 The native ZFS filesystem for Linux. Install the ubuntu-zfs package.
<output truncated>
gpg: key F6B0FC61: public key "Launchpad PPA for Native ZFS for Linux" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
OK
```

更新包列表。

```
$ sudo apt-get update
Ign http://us-west-2.ec2.archive.ubuntu.com trusty InRelease
Get:1 http://us-west-2.ec2.archive.ubuntu.com trusty-updates InRelease [64.4 kB]
<output truncated>
Fetched 10.3 MB in 4s (2,370 kB/s)
Reading package lists... Done
```

安装 ubuntu-zfs 包。

```
$ sudo apt-get install -y ubuntu-zfs
Reading package lists... Done
Building dependency tree
<output truncated>
```

加载 ZFS 模块，并检查是否加载成功。

```
$ sudo modprobe zfs
$ lsmod | grep zfs
zfs                  2768247  0
zunicode              331170  1 zfs
zcommon                55411  1 zfs
znvpair                89086  2 zfs,zcommon
spl                    96378  3 zfs,zcommon,znvpair
zavl                   15236  1 zfs
```

（4）创建 zpool。

```
$ sudo zpool create -f zpool-docker /dev/xvdb
$ sudo zfs list
NAME          USED  AVAIL    REFER  MOUNTPOINT
zpool-docker  55K   3.84G    19K    /zpool-docker
```

（5）挂载 ZFS 文件系统到/var/lib/docker。

```
$ sudo zfs create -o mountpoint=/var/lib/docker zpool-docker/docker
```

（6）检查 zpool 和挂载点。

```
$ sudo zfs list -t all
NAME                 USED  AVAIL  REFER  MOUNTPOINT
zpool-docker         93.5K  3.84G    19K  /zpool-docker
zpool-docker/docker  19K    3.84G    19K  /var/lib/docker
```

（7）在 Docker Daemon 中配置存储驱动。可以通过命令行启动 Dokcer Daemon。

```
$ sudo docker daemon --storage-driver=zfs&
```

也可以在 Docker Daemon 的配置文件中，配置存储驱动。

```
# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--storage-driver=zfs"
```

（8）启动 Docker Daemon。

```
$ sudo service docker start
docker start/running, process 2315
```

（9）检查 Docker Daemon。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: zfs
 Zpool: zpool-docker
 Zpool Health: ONLINE
 Parent Dataset: zpool-docker/docker
 Space Used By Parent: 27648
 Space Available: 4128139776
 Parent Quota: no
 Compression: off
Execution Driver: native-0.2
[...]
```

8.6.4　ZFS 的性能

下面几个因素会影响 ZFS 存储驱动的性能。

1. 内存

内存对 ZFS 的性能影响非常大。因为 ZFS 最初是为 SUN 的大机设计的，大机一般都使用很大的内存。在使用 ZFS 时，需要给宿主机配置很大的内存。

2. ZFS 开启的功能

使用 ZFS 的消重等高级功能，会占用非常多的内存。建议停止 ZFS 中的消重功能。

3. ZFS 缓存

ZFS 把存储设备中的数据缓存在内存中，叫作 ARC（Adaptive Replacement Cache）。ZFS 中的单一复制 ARC 功能允许多个 ZFS 克隆共享一个数据块。这样，在 ZFS 中，多个容器可以共享一个数据块。因此在 PaaS 的应用场景中，应该使用 ZFS 存储驱动。

4. 碎片

碎片是写时复制文件系统的一个副产品。ZFS 使用 128KB 的数据块，在写时复制时，会一次分配多个数据块，这样可以减少碎片。

5. 使用 ZFS 的原生驱动

尽管 ZFS 存储驱动支持 ZFS FUSE 方式，但是需要高性能时，不建议使用 ZFS FUSE。应该使用 ZFS 的原生驱动。

6. SSD

SSD 提供了高速的存储速度，为了获得高性能，可以使用 SSD 作为存储介质。

7. 使用数据卷

数据卷可以提供最好的数据读写性能。使用数据卷后，所有读写操作都会绕过存储驱动，不会增加额外开销。因此，在容器中，当需要大量写操作时，最好把这些数据保存在数据卷中。

8.7 Overlay 存储驱动

OverlayFS 是一种联合文件系统，与 AUFS 类似。从 Linux 内核 3.18 版本后，就进入了 Linux 内核源码主干。与 AUFS 相比，OverlayFS 更有优势，设计更简单，速度更快。

因为以上的优点，OverlayFS 在 Docker 社区中迅速流行了起来。很多人都认为可以使用 OverlayFS 取代 AUFS。OverlayFS 现在还不成熟，若要在生产环境中使用，需要首先在测试环境中进行测试。

Overlay 存储驱动使用了 OverlayFS 的一些特性管理镜像和容器。

从 Docker 1.12 版本开始，Docker 提供了 Overlay2 存储驱动，在 inode 使用方面，比 Overlay 存储驱动更有效。目前，只有 Linux 内核 4.0 及以上版本支持 Overlay2 存储驱动。

8.7.1 Overlay 中的镜像

OverlayFS 在 Linux 主机上，用到两个目录。两个目录使用分层结构，一个目录在下层，保存镜像层；另一个目录在上层，保存容器层。OverlayFS 把这两个目录组合起来，对外提供一个文件系统。OverlayFS 使用联合挂载技术组合这两个目录。在 OverlayFS 中，底层的目录叫作 lowerdir，顶层的目录叫作 upperdir，对外提供的统一文件系统叫作 merged。

图 8.25 展示了 Overlay 中，镜像和容器的存储方式。镜像层放在 lowerdir，容器层放在 uppderdir，容器挂载点放在 merged 中。

从图 8.25 可以看出，镜像层和容器层可以保存相同的文件。容器层中的文件会覆盖镜像层中的文件。

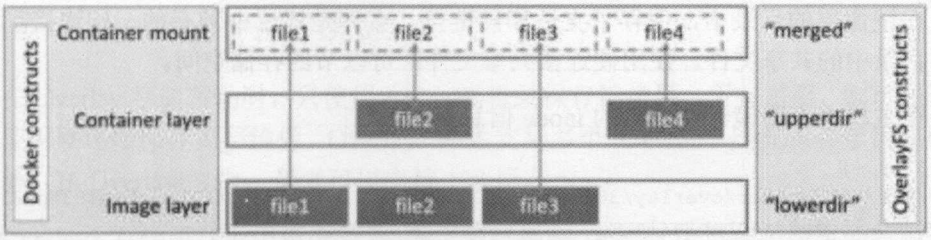


图 8.25　Overlay 中镜像和容器存储方式

Overlay 存储驱动只能使用两层，多层镜像不能在 OverlayFS 中映射为多个层。每个镜像层在/var/lib/docker/overlay 都有对应的目录，使用硬连接与底层关联数据。在 Docker 1.10 及后续版本中，镜像层 ID 与目录名字不一致。

运行容器时，存储驱动会把所有镜像层对应的目录组合起来，在上层添加一个容器层。镜像中最上面一层在 lowerdir 中，是只读的。容器层在 upperdir 中，是可读写的。

下面介绍一个 5 层镜像在 Overlay 模式下的结构。首先，下载 Ubuntu 最新镜像。

```
$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

每个镜像层在/var/lib/docker/overlay/都有对应的目录，其中保存了镜像层数据。

在/var/lib/docker/overlay/目录下，检查这 5 个镜像层和其中的数据。在 Docker 1.10 及后续版本中，镜像 ID 和子目录名不一致。

在主机上查看镜像层目录信息。

```
$ ls -l /var/lib/docker/overlay/
total 20
drwx------ 3 root root 4096 Jun 20 16:11
38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8
drwx------ 3 root root 4096 Jun 20 16:11
55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358
drwx------ 3 root root 4096 Jun 20 16:11
824c8a961a4f5e8fe4f4243dab57c5be798e7fd195f6d88ab06aea92ba931654
drwx------ 3 root root 4096 Jun 20 16:11
ad0fe55125ebf599da124da175174a4b8c1878afe6907bf7c78570341f308461
drwx------ 3 root root 4096 Jun 20 16:11
edab9b5e5bf73f2997524eebeac1de4cf9c8b904fa8ad3ec43b3504196aa3801
```

在镜像层的目录中，保存了文件和硬链接。文件是该层中独有的。硬连接指向下一层目录中的共享文件。使用硬连接共享文件，可以节约存储空间。

在主机上查看镜像层目录的 inode 信息。

```
$ ls -i
/var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72
e3e0a15baa8/root/bin/ls
19793696
/var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72
e3e0a15baa8/root/bin/ls
$ ls -i
/var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6f
f61aa0a5358/root/bin/ls
19793696
/var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6f
f61aa0a5358/root/bin/ls
```

容器层同样保存在/var/lib/docker/overlay/目录下。在运行容器的容器层目录中可以发现这些文件。

在主机上查看容器层信息。

```
$ ls -l /var/lib/docker/overlay/<directory-of-running-container>
total 16
-rw-r--r-- 1 root root   64 Jun 20 16:39 lower-id
drwxr-xr-x 1 root root 4096 Jun 20 16:39 merged
drwxr-xr-x 4 root root 4096 Jun 20 16:39 upper
drwx------ 3 root root 4096 Jun 20 16:39 work
```

这四个文件是 OverlayFS 中的 artifact。lower-id 文件包含了镜像顶层的 ID。镜像顶层保存在 lowerdir 中。

查看 lower-id 文件信息。

```
$ cat
/var/lib/docker/overlay/ec444863a55a9f1ca2df72223d459c5d940a721b2288ff86a3f27
be28b53be6c/lower-id
55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358
```

容器层保存在 upper 目录中，容器中修改的数据都保存在该目录中。

merged 目录是容器的挂载点，合并了 lowerdir 和 upperdir。

work 目录是 OverlayFS 使用的，一些 OverlayFS 的操作需要使用该目录，如 copy_up。

可以使用 mount 命令查看挂载关系。

```
$ mount | grep overlay
overlay on /var/lib/docker/overlay/ec444863a55a.../merged
type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay/55f1e14c361b.../root,
upperdir=/var/lib/docker/overlay/ec444863a55a.../upper,
workdir=/var/lib/docker/overlay/ec444863a55a.../work)
```

8.7.2　Overlay2 中的镜像

在 Overlay 存储驱动时，只使用 lowerdir 目录保存镜像层，所有需要，则可在该目录中使用硬连接保存多层镜像。Overlay2 原生支持多个 lowerdir，因此保存多层镜像更有优势。在 OverlayFS 中，最多可以支持 128 层。

Overlay2 存储驱动使用更少的 inode，可以提高 Docker 中层相关命令的效率，如 docker build、docker commit 等。

Overlay2 中，镜像和容器保存在/var/lib/docker/overlay2 目录下。下面介绍一个 5 层镜像在 Overlay2 中的结构。

查看主机上的镜像层信息。

```
$ ls -l /var/lib/docker/overlay2
total 24
drwx------ 5 root root 4096 Jun 20 07:36
223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7
drwx------ 3 root root 4096 Jun 20 07:36
3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b
drwx------ 5 root root 4096 Jun 20 07:36
4e9fa83caff3e8f4cc83693fa407a4a9fac9573deaf481506c102d484dd1e6a1
drwx------ 5 root root 4096 Jun 20 07:36
e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bdff3f59cff5
drwx------ 5 root root 4096 Jun 20 07:36
eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda7422fed78afed
drwx------ 2 root root 4096 Jun 20 07:36 l
```

/var/lib/docker/overlay2/中的 l 目录保存了镜像层的短标识，每个短标识都是一个符号连接。短标识用于解决 mount 参数中长字符串超过页大小限制问题。

查看镜像层的短标识。

```
$ ls -l /var/lib/docker/overlay2/l
total 20
lrwxrwxrwx 1 root root 72 Jun 20 07:36 6Y5IM2XC7TSNIJZZFLJCS6I4I4
-> ../3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/diff
lrwxrwxrwx 1 root root 72 Jun 20 07:36 B3WWEFKBG3PLLV737KZFIASSW7
-> ../4e9fa83caff3e8f4cc83693fa407a4a9fac9573deaf481506c102d484dd1e6a1/diff
lrwxrwxrwx 1 root root 72 Jun 20 07:36 JEYMODZYFCZFYSDABYXD5MF6YO
-> ../eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda7422fed78afed/diff
lrwxrwxrwx 1 root root 72 Jun 20 07:36 NFYKDW6APBCCUCTOUSYDH4DXATd
-> ../223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7/diff
lrwxrwxrwx 1 root root 72 Jun 20 07:36 UL2MW33MSE3Q5VYIKBRN4ZAGQP
-> ../e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bdff3f59cff5/diff
```

最底层的镜像层包含了 link 文件，其中保存了短标识符的名字。diff 目录包含了镜像层的内容。

查看镜像层目录的 link 文件。

```
$ ls
/var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e
506e494e348b/
diff  link
$ cat
/var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e
506e494e348b/link
6Y5IM2XC7TSNIJZZFLJCS6I4I4
$ ls     /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0
f87905b2e     506e494e348b/diff
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
```

倒数第二个镜像层包含了 lower 文件，保存镜像层的组成信息。diff 目录保存镜像层内容，同时包含了 merged 和 work 两个目录。

查看镜像层目录的 lower 文件和 diff 目录。

```
$ ls     /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d05
0924fd1bf     cdb278b866f7
diff  link  lower  merged  work
$ cat    /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d05
0924fd1bf     cdb278b866f7/lower
l/6Y5IM2XC7TSNIJZZFLJCS6I4I4
$ ls     /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d05
0924fd1bf     cdb278b866f7/diff/
etc  sbin  usr  var
```

容器层也是类似的目录结构。lower 文件中的镜像层使用:分隔。在文件中，镜像层按顺序排列，排在前面的镜像层放在上面，排在后面的镜像层放在下面。

查看容器层目录的 lower 文件。

```
$ ls -l /var/lib/docker/overlay/<directory-of-running-container>
$ cat /var/lib/docker/overlay/<directory-of-running-container>/lower
l/DJA75GUWHWG7EWICFYX54FIOVT:l/B3WWEFKBG3PLLV737KZFIASSW7:l/JEYMODZYFCZFYSDAB
YXD5MF6YO:l/UL2MW33MSE3Q5VYIKBRN4ZAGQP:l/NFYKDW6APBCCUCTOUSYDH4DXAT:l/6Y5IM2X
C7TSNIJZZFLJCS6I4I4
```

使用 mount 命令可以查看挂载结构。

```
$ mount | grep overlay
overlay on
/var/lib/docker/overlay2/9186877cdf386d0a3b016149cf30c208f326dca307529e646afc
e5b3f83f5304/merged
type overlay (rw,relatime,
lowerdir=l/DJA75GUWHWG7EWICFYX54FIOVT:l/B3WWEFKBG3PLLV737KZFIASSW7:l/JEYMODZY
FCZFYSDABYXD5MF6YO:l/UL2MW33MSE3Q5VYIKBRN4ZAGQP:l/NFYKDW6APBCCUCTOUSYDH4DXAT:
l/6Y5IM2XC7TSNIJZZFLJCS6I4I4,
```

```
upperdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/diff,
workdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/work)
```

8.7.3　Overlay 中的读写

1. 在容器中读取文件

在容器中读取文件分为不同场景。

（1）目标文件不在容器层中，Overlay 会从镜像层读取文件。此时，对容器性能的影响很小。

（2）目标文件只在容器层中，Overlay 直接从容器层读取文件。

（3）目标文件在容器层和镜像层同时存在，overlay 读入容器层中的文件。此时，容器层的文件会覆盖镜像层的文件。

2. 在容器中修改文件

在容器中第一次修改文件，此时文件不在容器层中。Overlay/Overlay2 存储驱动会把文件从镜像层复制到容器层。所有文件中的修改都保存在容器层中。

OverlayFS 文件系统工作在文件层而不是数据块层。复制文件时，会复制整个文件。在大文件中仅修改了很小的数据块，也需要复制整个文件。这会对容器的写性能造成非常大的影响。以下两点值得注意。

（1）只是在第一次修改文件时，需要把文件从镜像层复制到容器层。后续操作都是在容器层中完成。

（2）OverlayFS 只有两层：lowerdir 和 upperdir。在很深的目录树中，搜索文件时，Overlay 比 AUFS 速度更快。

3. 在容器中删除文件和目录

在容器中删除文件时，Overlay 存储驱动在容器层中新建一个 without 文件，该文件用于隐藏镜像层中的目标文件。在容器层中删除目录时，Overlay 存储驱动在容器层中新建一个 opaque 目录，该目录用于隐藏镜像层中的目标目录。Overlay 存储驱动不会删除镜像层中的目标文件和目标目录。

8.7.4　如何配置 Overlay/Overlay2

若使用 Overlay 存储驱动，则要求宿主机的 Linux 内核必须是 3.18 版本。若使用 Overlay2 存储驱动，则要求宿主机的 Linux 内核必须是 4.0 或以上版本。OverlayFS 可以工作在大部分的 Linux 文件系统上，推荐在生产环境中使用 ext4 文件系统。

（1）在宿主机上停止 Docker Daemon。

```
$ service docker stop
```

（2）检查 Linux 内核版本，并加载 Overlay 模块。

```
$ uname -r
3.19.0-21-generic

$ lsmod | grep overlay
overlay
```

（3）在 Docker Daemon 中配置存储驱动。可以通过命令行启动 Dokcer Daemon。

```
$ sudo docker daemon --storage-driver=overlay&
```

也可以在 Docker Daemon 的配置文件中，配置存储驱动。

```
# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--storage-driver=overlay"
```

（4）使用 docker info 命令检查 Overlay 是否生效。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: overlay
[...]
```

使用 Overlay/Overlay2 存储驱动以后，Docker 会自动创建 Overlay 挂载点，并在其中创建 lowerdir、upperdir、merged 和 workdir 等目录。

8.7.5　Overlay 的性能

总体来讲，Overlay/Overlay2 存储驱动很快，比 AUFS 和 Devicemapper 都要快。在某些场景下，甚至快于 Btrfs。

1．页缓存

OverlayFS 支持共享页缓存。当多个容器访问同一个文件时，可以共享一个页缓存。Overlay/Overlay2 存储驱动在内存使用上非常有效。Overlay/Overlay2 适用于 PaaS 环境。

2．从镜像层复制文件到容器层

与 AUFS 一样，在容器中第一次修改文件时，Overlay/Overlay2 存储驱动会把文件从镜像层复制到容器层，这样会影响写操作的性能，特别是当文件很大时。一旦完成复制，后续的操作都在容器层完成。

OverlayFS 的复制操作比 AUFS 快。因为 AUFS 中有更多的层，在 AUFS 中，搜索很深的目录树会产生很大的开销。

3．YUM 和安装包

OverlayFS 没有完全支持 POSIX 协议，只支持其中的一些接口。OverlayFS 中的某些操作会破坏 POSIX 协议，如从镜像层复制文件到容器层的操作。若在宿主机上不添加补救措施，则可能导致容器中的 yum 命令执行失败。

4. inode 限制

Overlay 存储驱动会消耗大量的 inode，特别是当宿主机上的镜像层和容器层迅速增长时。在宿主机上，如果有大量的镜像，启停容器会迅速消耗 inode。而 overlay2 存储驱动就不会有这些问题。

开发者只能在创建文件系统时修改 inode 数量。一旦建立了文件系统，就不能修改 inode 数量。使用 Overlay 作为存储驱动时，建议使用单独的裸设备存储镜像和容器。在该设备上创建文件系统并修改 inode 数量，把该设备挂载到/var/lib/docker 上。

5. SSD

SSD 提供了高速的存储速度，为了获得高性能，可以使用 SSD 作为存储介质。

6. 使用数据卷

数据卷可以提供最好的数据读写性能。使用数据卷后，所有读写操作都会绕过存储驱动，不会增加额外开销。因此，在容器中，当需要大量写操作时，最好把这些数据保存在数据卷中。

8.8 习题

本章详细介绍了 Docker 中的各种存储驱动。接下来，通过习题和实验检验本章的学习成果。

（1）把 Docker 的存储驱动修改为 AUFS，下载 httpd 镜像，并启动容器。检查/var/lib/docker/aufs 目录中的内容。

（2）把 Docker 的存储驱动修改为 Devicemapper，下载 httpd 镜像，并启动容器。检查/var/lib/docker/devicemapper 目录中的内容。

（3）把 Docker 的存储驱动修改为 Btrfs，下载 httpd 镜像，并启动容器。检查/var/lib/docker/btrfs 目录中的内容。

（4）把 Docker 的存储驱动修改为 ZFS，下载 httpd 镜像，并启动容器。检查/var/lib/docker/zfs 目录中的内容。

（5）把 Docker 的存储驱动修改为 Overlay，下载 httpd 镜像，并启动容器。检查/var/lib/docker/overlay 目录中的内容。

把 Docker 的存储驱动修改为 Overlay2，下载 httpd 镜像，并启动容器。检查/var/lib/docker/overlay 目录中的内容。

第9章　定制 Docker Daemon

Docker Daemon 是一个守护进程，Docker Client 通过命令行与 Docker Daemon 通信，完成 Docker 相关操作。Docker Daemon 有近 50 个启动选项，配置这些选项，可以提高 Docker 的运行效率，增加 Docker 的安全性。定制适合自己应用场景的 Docker 运行环境，Docker Daemon 的选项包括仓库、安全、日志、存储、网络等。

9.1　修改 Docker Daemon 的三种方式

修改 Docker Daemon 有三种方式，分别是通过命令行修改、修改启动项和修改配置文件。这三种方式使用在不同的场景中。

（1）当需要研究 Docker Daemon 的不同选项，需要尝试不同选项组合时，可以使用命令行的方式。此时，Docker Daemon 运行在前端，日志直接打印在终端上，方便调试。

（2）在生产环境中，应使用后面两种方式。当 Docker Daemon 的配置很稳定，长时间不需要修改时，可以使用把 Docker Daemon 的选项放在在启动项中的方式。当 Docker Daemon 的配置需要定时修改时，可以把固定的选项放在启动项中，变化的选项放在配置文件中，具体见表 9.1。

表 9.1　Docker Daemon 的配置项

配　置　项	说　明
--api-cors-header	设置远程 API 中的 CORS 头
--authorization-plugin=[]	设置认证插件列表
-b, --bridge	设置容器使用的网桥
--bip	设置 Docker0 的 IP 和子网掩码
--cgroup-parent	设置容器的父 CGroups
--cluster-advertise	设置加入集群时，使用的 IP 或网口
--cluster-store	设置集群使用的存储

续表

配　置　项	说　明
--cluster-store-opt=map[]	设置集群使用的存储参数
--config-file=/etc/docker/daemon.json	设置 Docker Daemon 使用的配置文件路径
-D, --debug	开启 Docker Daemon 的调试模式
--default-gateway	设置容器使用的 IPv4 网关地址
--default-gateway-v6	设置容器使用的 IPv6 网关地址
--default-ulimit=[]	设置容器中的默认 ulimit 值
--disable-legacy-registry	禁止从旧版本的镜像仓库下载镜像。目前的仓库版本为 V2，设置该参数后，就不能从 V1 版本的仓库下载镜像
--dns=[]	设置容器使用的 DNS Server 列表
--dns-opt=[]	设置容器使用的 DNS 选项
--dns-search=[]	设置容器使用的 DNS 默认域名
--exec-opt=[]	设置 Docker Daemon 管理容器使用的参数
--exec-root=/var/run/docker	设置 execdriver 的根目录
--fixed-cidr	设置容器使用的 IPv4 地址范围
--fixed-cidr-v6	设置容器使用的 IPv6 地址范围
-G, --group=docker	设置 UNIX Socket 的用户组
-g, --graph=/var/lib/docker	设置 Docker Daemon 运行时的根目录
-H, --host=[]	设置 Docker Daemon 监听的 IP 和端口
--help	帮助文档
--icc=true	启动内联容器通信
--insecure-registry=[]	设置非安全镜像仓库列表
--ip=0.0.0.0	设置导出容器端口时，使用的 IP 地址
--ip-forward=true	开启系统的 net.ipv4.ip_forward
--ip-masq=true	开启 IP 伪装
--iptables=true	开启自动更新 iptables 规则
--ipv6	启动 IPv6 网络
-l, --log-level=info	设置日志等级
--label=[]	设置 key=value 标签
--log-driver=json-file	设置容器日志驱动
--log-opt=map[]	设置容器日志驱动参数
--mtu	设置容器的 MTU
-p, --pidfile=/var/run/docker.pid	设置 Docker Daemon 使用的 PID 文件路径
--registry-mirror=[]	设置 mirro 镜像仓库列表
-s, --storage-driver	设置 Docker Daemon 保存镜像容器的方式
--selinux-enabled	开启 SELinux
--storage-opt=[]	设置不同--storage-driver 的参数
--tls	开启 TLS
--tlscacert=~/.docker/ca.pem	用于签名的 CA 证书
--tlscert=~/.docker/cert.pem	TLS 使用的证书
--tlskey=~/.docker/key.pem	TLS 使用的私钥
--tlsverify	开启 TLS，并对远端的 Docker Client 进行认证
--userland-proxy=true	为 loopback 通信使用 userland 代理
--userns-remap	设置 namespaces 的用户和用户组

9.1.1　直接启动 Docker Daemon

在配置 Docker Daemon 时，可以通过命令行启动 Docker Daemon。此时，所有日志打印在终端上，方便调试。可以在命令行后追加配置项，这些配置项会立即生效。

在命令行中启动 Docker Daemon。

```
# docker daemon
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewalld running: false
INFO[0001] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16.
Daemon option --bip can be used to set a preferred IP address
INFO[0001] Loading containers: start.
.....
INFO[0001] Loading containers: done.
INFO[0001] Daemon has completed initialization
INFO[0001] Docker daemon commit=20f81dd execdriver=native-0.2 graphdriver=aufs
version=1.10.3
INFO[0001] API listen on /var/run/docker.sock
...
```

通常，可以在命令行中添加的选项具体见表 9.2。

表 9.2　命令行中添加的选项

配　置　项	功　能
-D，--debug=false	开启或关闭调试模式，默认情况下关闭调试模式
-H，--host=[]	设置 Docker Daemon 的监听方式
--tls=false	开启或关闭 TLS，默认情况下关闭 TLS

在命令行中启动 Docker Daemon 时，可以设置一些启动项。

```
# docker daemon -D --tls=true --tlscacert=/root/docker/ca.crt
--tlscert=/root/docker/server.crt --tlskey=/root/docker/server.key -H tcp://127.0.0.1:2376
```

9.1.2　修改 Docker Daemon 启动项

Docker Daemon 的启动项存放在启动文件中。Docker 启动时，会从启动文件中读取启动项。在不同操作系统中，启动文件的路径和格式都不相同。在 Linux 系统下，通常使用 SysV Init、Upstart 或 Systemd 管理进程。在不同发行版中管理 Docker Daemon，则需要使用对应的进程管理工具管理 Docker Daemon。

1. Ubuntu

Ubuntu 使用 Upstart 管理进程，默认情况下，Docker 的启动脚本是/etc/init/docker.conf。

在 upstart 中，使用以下命令管理 Docker Daemon。

```
# start docker
```

```
# stop docker
# restart docker
# status docker
```

Docker Daemon 的启动文件是/etc/default/docker，修改 Docker Daemon 启动项时，需要修改 DOCKER_OPTS 的值。

（1）修改 Docker Daemon 的启动文件，需要首先获得 root 权限。可以以 root 用户登录，其他用户使用 sudo 获得 root 权限。

（2）检查是否存在/etc/default/docker 文件，如果没有，则需要手动创建该文件。

```
# touch /etc/default/docker
```

（3）编辑/etc/default/docker 文件，修改 DOCKER_OPTS，加入 Docker Daemon 启动项。

```
# vi /etc/default/docker
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--tlsverify --tlscacert=/root/docker/ca.crt
--tlscert=/root/docker/server.crt --tlskey=/root/docker/server.key
--registry-mirror=http://mirror.ghostcloud.cn"

# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

（4）重启 Docker Daemon，使配置生效。

```
# restart docker
```

（5）查询配置选项是否生效。

```
# ps -ef | grep docker
root     31474     1  0 Apr12 ?        00:09:48 /usr/bin/docker daemon
--insecure-registry=hub.ghostcloud.cn
```

（6）Ubuntu 中，Docker Daemon 的日志文是/var/log/upstart/docker.log。通过该文件，可以查看 Docker Daemon 的日志。

```
# tailf /var/log/upstart/docker.log
INFO[0000] Loading containers: done.
INFO[0000] Docker daemon commit=1b09a95-unsupported graphdriver=aufs version=1.11.0-dev
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
```

2. Cent OS

Cent OS 7 以后的版本使用 systemd 管理服务。在 systemd 中，使用以下命令管理 Docker Daemon。

```
# systemctl start docker
# systemctl stop docker
# systemctl restart docker
# systemctl status docker
```

也可以通过 systemd 把 Docker Daemon 加入自启动。

```
# systemctl enable docker
```

Docker Daemon 的启动文件放在/etc/systemd/system/docker.service.d 目录中。如果没有这个目录，则需要手工建立该目录。在该目录中，创建 docker.conf 的配置文件，修改 ExecStart 参数。

（1）修改 Docker Daemon 的配置，需要首先获得 root 权限。可以以 root 用户登录，其他用户使用 sudo 获得 root 权限。

（2）创建/etc/systemd/system/docker.service.d 目录。

```
# mkdir /etc/systemd/system/docker.service.d
```

（3）在/etc/systemd/system/docker.service.d 目录下，创建 docker.conf 文件，保存 Docker Daemon 的启动项。

（4）修改 docker.conf 文件，添加启动项。注意，修改 ExecStart 参数时，必须配置两行 ExecStart 参数。第一行为 ExecStart=；第二行加入 Docker Daemon 的启动项。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// --tlsverify --tlscacert=/root/.ghostcloud/
conf/ca.crt --tlscert=/root/.ghostcloud/conf/server.crt --tlskey=/root/.ghostcloud/
conf/server.key
-H=tcp://127.0.0.1:2376 --registry-mirror=http://mirror.ghostcloud.cn
--insecure-registry=hub.ghostcloud.cn --storage-driver=devicemapper --storage-opt
dm.thinpooldev=/dev/mapper/gc--docker--vg-gc--docker--thin--pool --storage-opt
dm.basesize=20GB --storage-opt dm.fs=xfs --storage-opt dm.use_deferred_removal=true
--storage-opt dm.use_deferred_deletion=true
```

（5）配置完毕，加载配置文件，重启 Docker Daemon。

```
# systemctl daemon-reload
# systemctl start docker
```

（6）检查配置是否生效。

```
# ps -ef | grep docker
root    18482    1  0 4月18 ?      00:20:17 /usr/bin/docker daemon -H fd:// --tlsverify
--tlscacert=/root/.ghostcloud/conf/ca.crt--tlscert=/root/.ghostcloud/conf/server.crt
--tlskey=/root/.ghostcloud/conf/server.key -H=tcp://127.0.0.1:2376
--registry-mirror=http://mirror.ghostcloud.cn --insecure-registry=hub.ghostcloud.cn
```

```
--storage-driver=devicemapper --storage-opt
dm.thinpooldev=/dev/mapper/gc--docker--vg-gc--docker--thin--pool --storage-opt
dm.basesize=20GB --storage-opt dm.fs=xfs --storage-opt dm.use_deferred_removal=true
--storage-opt dm.use_deferred_deletion=true
```

（7）systemd 使用 journal 程序管理日志，查看 Dokcer Daemon 日志，需要使用 journalctl –u docker 命令。

```
# journalctl -u docker
May 06 00:22:05 localhost.localdomain systemd[1]: Starting Docker Application
Container Engine...
May 06 00:22:05 localhost.localdomain docker[2495]: time="2015-05-06T00:22:05Z"
level="info" msg="+job serveapi(unix:///var/run/docker.sock)"
May 06 00:22:05 localhost.localdomain docker[2495]: time="2015-05-06T00:22:05Z"
level="info" msg="Listening for HTTP on unix (/var/run/docker.sock)"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="+job init_networkdriver()"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="-job init_networkdriver() = OK (0)"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="Loading containers: start."
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="Loading containers: done."
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="Docker daemon commit=1b09a95-unsupported graphdriver=aufs
version=1.11.0-dev"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z"
level="info" msg="+job acceptconnections()"
```

9.1.3　自定义 Docker Daemon 配置文件

Docker Daemon 启动时，首先会从启动文件中读取启动项，接下来会追加配置文件中配置项。通过--config-file=/etc/docker/daemon.json 设置 Docker Daemon 的配置文件路径。

编辑/etc/docker/daemon.json 文件，设置 Docker Daemon 的配置。配置文件中的配置项不能与 9.1.2 节中配置的 Docker Daemon 启动项重复，一旦重复，Docker Daemon 就不能启动。Docker Daemon 会判断配置文件中的配置项是否出现在 Docker Daemon 的启动项中，如果配置项重复出现，则 Docker Daemon 启动时就会报错。

```
# vi /etc/docker/daemon.json
{
    "registry-mirror":"http://mirror.ghostcloud.cn"
}
# docker daemon -H tcp://0.0.0.0:2375 --registry-mirror=http://mirror.ghostcloud.com
unable to configure the Docker daemon with file /etc/docker/daemon.json: the
following directives are specified both as a flag and in the configuration file:
registry-mirror: (from flag: [http://mirror.ghostcloud.com/], from file:
http://mirror.ghostcloud.cn)
```

在 Linux 中，默认配置文件为/etc/docker/daemon.json；在 Windows 中，默认配置文件为%programdata%\docker\config\daemon.json。配置文件为 JSON 格式，参数名称与 Docker Daemon 配置项相同。例如，Docker Daemon --registry-mirror=http://mirror.ghostcloud.cn 对应的参数为"registry-mirror":"http://mirror.ghostcloud.cn"。对于支持多个值的配置项，需要使用配置名称的复数形式。例如，配置项--label，在配置文件中，变为 labels。配置文件中没有出现的配置项会被忽略。

不是所有的启动项都可以写在配置文件中，下面的配置文件，包含了允许出现的所有配置项。

```
{
    "authorization-plugins": [],
    "dns": [],
    "dns-opts": [],
    "dns-search": [],
    "exec-opts": [],
    "exec-root": "",
    "storage-driver": "",
    "storage-opts": [],
    "labels": [],
    "log-driver": "",
    "log-opts": [],
    "mtu": 0,
    "pidfile": "",
    "graph": "",
    "cluster-store": "",
    "cluster-store-opts": [],
    "cluster-advertise": "",
    "debug": true,
    "hosts": [],
    "log-level": "",
    "tls": true,
    "tlsverify": true,
    "tlscacert": "",
    "tlscert": "",
    "tlskey": "",
    "api-cors-headers": "",
    "selinux-enabled": false,
    "userns-remap": "",
    "group": "",
    "cgroup-parent": "",
    "default-ulimits": {},
    "ipv6": false,
    "iptables": false,
    "ip-forward": false,
    "ip-mask": false,
    "userland-proxy": false,
    "ip": "0.0.0.0",
    "bridge": "",
```

```
    "bip": "",
    "fixed-cidr": "",
    "fixed-cidr-v6": "",
    "default-gateway": "",
    "default-gateway-v6": "",
    "icc": false,
    "raw-logs": false,
    "registry-mirrors": [],
    "insecure-registries": [],
    "disable-legacy-registry": false
}
```

某些配置项支持动态加载，不需要重启 Docker Daemon，就可以更新配置项。在 Linux 中，通过发送信号量实现动态加载。在 Windows 中，通过发送 event 给 Global\docker-daemon-config-$PID，实现动态加载。更新配置项时，必须避免与启动项重复。

以下是支持动态更改的配置项。

- debug——设置为 true 时，启动调试模式。
- cluster-store——更新集群中分布式存储后端的地址。
- cluster-store-opts——更新集群选项。
- cluster-advertise——更新 Docker Daemon 在集群中的 IP 地址。
- labels——修改 Docker Daemon 中的标签。

更新集群相关的配置时，有特殊限制：要求更新前，没有配置过相关参数。例如，要更新--cluster-store，要求之前的启动项和配置文件中，都没有配置--cluster-store。如果更改了集群相关的配置，日志中会记录一条警告信息。

9.2　仓库相关配置

9.2.1　--disable-legacy-registry 选项

设置不从旧版本的镜像仓库下载镜像。Docker 从 1.6 版本后，就支持从 V2 版的镜像仓库下载镜像。为了向下兼容，Docker 会首先尝试连接镜像仓库中的 V2 接口，如果失败，则会再尝试镜像仓库中的 V1 接口。设置该参数后，Docker 就不会连接镜像仓库中的 V1 接口。

本书通过一个小实验，了解 Docker 下载镜像的流程。找一台需要认证的镜像仓库，不登录，尝试从认证容器下载镜像。此时，Docker 会报错，并把调试信息打印出来。

第一次启动 Docker Daemon 时不加入--disable-legacy-registry 参数。

```
# docker daemon
```

```
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
```

从日志中可以看出，Docker 首先尝试 V2 接口，失败以后，会尝试 V1 接口。

```
# docker pull hub.ghostcloud.cn/ubuntu
Using default tag: latest
Error response from daemon: unable to ping registry endpoint https://
hub.ghostcloud.cn/v0/
v2 ping attempt failed with error: Get https://hub.ghostcloud.cn/v2/: dial tcp
120.27.143.61:443: getsockopt: connection refused
 v1 ping attempt failed with error: Get https://hub.ghostcloud.cn/v1/_ping: dial
 tcp 120.27.143.61:443: getsockopt: connection refused
```

第二次启动 Docker Daemon 时加入--disable-legacy-registry。

```
# docker daemon --disable-legacy-registry
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
```

此时，Docker 只会尝试 V2 接口，如果失败，则立即退出。

```
# docker pull hub.ghostcloud.cn/ubuntu
Using default tag: latest
Error response from daemon: Get https://hub.ghostcloud.cn/v2/: dial tcp
120.27.143.61:443: getsockopt: connection refused
```

9.2.2　--registry-mirror 选项

在国内，使用 Docker 仓库最大的问题就是网速问题。用户常常不能从 Docker Hub 上下载到丰富的镜像资源。要么就是不能连上，要么就是下载缓慢。为了解决这个问题，Docker Daemon 提供了镜像仓库的选项。只要配置好镜像仓库，用户就能体会到飞一般的下载速度。

镜像仓库会自动同步 Docker Hub 上的镜像到本地，为用户在国内建立一个镜像缓存。用户通过镜像仓库下载镜像时，不用访问 Docker Hub，可以直接通过国内的镜像仓库下载镜像。目前，国内已经有一些免费的镜像仓库，如 http://mirror.ghostcloud.cn，读者可以使用。

配置时，可以设置多个镜像仓库。

```
# docker daemon --registry-mirror=http://mirror.ghostcloud.cn
--registry-mirror=http://mirror2.company.com
```

9.2.3　--insecure-registry 选项

Docker 把镜像仓库分为安全的和不安全的。从不安全的镜像仓库下载镜像、上传镜像或搜索镜像，都会失败。从不安全的镜像仓库下载镜像，需要在 Docker Daemon 中添加—insecure-registry。

安全的镜像仓库要使用 TLS，并且主机上保存了该镜像仓库的 CA 证书。例如，https://secure1.registry.com:5000 是一个安全的镜像仓库，因为这个镜像仓库使用了 HTTPS 协议，并且在主机上保存了/etc/docker/certs.d/secure1.registry.com:5000/ca.crt 的证书。

不安全的镜像仓库可能是没有使用 TLS，如 http://insecur.registry.com。或者镜像仓库使用了 TLS，但是主机上没有保存 CA 证书。例如，主机上没有保存/etc/docker/certs.d/secure2.registry.com:5000/ca.crt 证书，https://secure2.registry.com:5000 会被标识为不安全的镜像仓库。

用户建立自己的私有镜像仓库，可以保证镜像仓库是安全的，一般不会设置 TLS。如果从这些镜像仓库下载镜像，需要把镜像仓库添加到—insecure-registry 列表中。

添加仓库时，有以下两种格式。

（1）通过域名+端口的方式添加单独的镜像仓库。

```
--insecure-registry myregistry:5000
```

（2）通过网段的方式，批量添加镜像仓库。Docker Daemon 会认为 10.1.0.0/16 网段中的镜像仓库都是安全的。

```
--insecure-registry 10.1.0.0/16
```

9.3　安全相关配置

下面的配置是 Docker Daemon 中和安全相关的配置项。

9.3.1　-p, --pidfile 选项

设置 Docker Daemon 使用的 pid 文件，默认为/var/run/docker.pid。

9.3.2　-H, --host 选项

Docker Daemon 作为守护进程在后台运行，接收 Docker Client 发送的指命令。Docker Daemon 与 Docker Client 通过 Socket 方式通信，Docker Daemon 监听 Socket 有三种方式，分别是 unix、tcp 和 fd。

默认情况下，Docker Daemon 与 Docker Client 在同一台主机上，使用本地 Socket 通信。Docker Daemon 的配置为 unix:///var/run/docker.sock。此时，要求运行 Docker 命令的用户具有 root 权限，或者已经加入了 Docker 用户组。

可以通过下面的选项配置 Docker Daemon 监听的 IP 和端口。

```
-H, --host=[]
```

如果 Docker Daemon 与 Docker Client 不在同一台主机上，Docker Client 采用远端通信的方式向 Docker Daemon 发送 Docker 命令。此时，需要在 Docker Daemon 中开启 TCP Socket。如果没有启动--tls 选项，则 Docker Daemon 与 Docker Client 的通信没有经过认证和加密。Docker Daemon 可以接收任何 Docker Client 发送的命令。下节将介绍如何通过--tls 对远端通信进行认证和加密。在 Docker 中规定，Docker Daemon 使用 2375 作为非加密端口，使用 2376 作为加密端口。

当主机有多个 IP 时，可以设置 Docker Daemon 在某个 IP 上监听。设置-H tcp://0.0.0.0:2375 时，Docker Daemon 在主机的所有 IP 上监听，端口为 2375。

使用 ifconfig 查看主机网络信息，该主机有两块网卡。

```
# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
       inet 192.168.0.1  netmask 255.255.240.0  broadcast 0.0.0.0
       ……

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
       inet 10.35.21.8  netmask 255.255.252.0  broadcast 10.24.207.255
       ……

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
       inet 120.135.21.15  netmask 255.255.252.0  broadcast 120.25.95.255
```

启动 Docker Daemon，同时在两块网卡上监听。

```
# docker daemon -H tcp://0.0.0.0:2375
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on [::]:2375
```

此时，Docker Client 可以向主机的任意 IP 发送命令。

```
# docker -H 10.35.21.8:2375 info
Containers: 3
 Running: 0
 Paused: 0
 Stopped: 3
Images: 12
Server Version: 1.10.3
……
# docker -H 120.135.21.15:2375 info
```

```
Containers: 3
 Running: 0
 Paused: 0
 Stopped: 3
Images: 12
Server Version: 1.10.3
……
```

也可以设置 Docker Daemmon 在某一个 IP 上监听。如 docker daemon -H tcp://10.35.21.8:2375。

```
# docker daemon -H tcp://10.35.21.8:2375
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on 10.35.21.8:2375
```

Docker Client 向 10.35.21.8:2375 发送命令，成功。

```
# docker -H 10.35.21.8:2375 info
Containers: 3
 Running: 0
 Paused: 0
 Stopped: 3
Images: 12
Server Version: 1.10.3
……
```

Docker Client 向另外一个 IP 发送命令，失败。

```
# docker -H 120.135.21.15:2375 info
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

在使用 systemd 的系统中，Docker Daemon 和 Docker Client 使用 fd 方式通信。例如，docker daemon -H fd://。

9.3.3　--tls, --tlscacert, --tlscert, --tlskey, --tlsverify 选项

默认情况下，Docker Client 与 Docker Daemon 在同一台主机上，两者通过 UNIX Socket 文件的方式通信。Docker Daemon 也可以接收远端主机上 Docker Client 发送的指令。用户可以通过 Docker 命令行或者调用函数库操作远程主机的 Docker Daemon。使用远程通信需要使用 TLS 进行安全加密。

使用 TLS 通信，需要在 Docker Daemon 和 Docker Client 中加入 TLS 相关的配置。首先创建一套 CA 证书，然后在 Docker Daemon 和 Docker Client 中同时开启 TLS 认证，并分别添加服务器 CA 证书和客户端 CA 证书。此时，Docker Daemon 只接收通过 CA 认证的客户端发送的命令；Docker Client 也只能向通过 CA 认证的服务端发送指令。

```
--tls                              Use TLS; implied by --tlsverify
--tlscacert="~/.docker/ca.pem"     Trust certs signed only by this CA
--tlscert="~/.docker/cert.pem"     Path to TLS certificate file
--tlskey="~/.docker/key.pem"       Path to TLS key file
--tlsverify                        Use TLS and verify the remote
```

使用 TLS 通信，需要创建 3 个证书文件和 2 个秘钥文件，分别是 CA 证书、服务器证书、客户端证书、服务器公钥和客户端公钥。在 Docker Daemon 的服务器上，需要使用 CA 证书、服务器证书和服务器公钥。在 Docker Client 的服务器上，需要使用 CA 证书、客户端证书和客户端公钥。生成这些证书和秘钥的过程比较复杂，下面将详细介绍如何逐步创建这些文件。

在本例中将展示如何在 Linux 环境下，创建 CA 证书、服务器秘钥、客户端秘钥，以及开启 Docker Daemon 和 Docker Client 的 TLS 通信。Mac OS 使用了不同的 OpenSSL 生成证书，因此本例不适用于 Mac OS。

（1）创建 RSA 私钥，使用该私钥对 CA 证书签名。-out ca.key 是 RSA 私钥的文件名。4096 是私钥的 bit 数。默认情况下，私钥长度为 512bit。在命令行提示中，输入 RSA 私钥的密码。

```
# openssl genrsa -aes256 -out ca.key 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................................
.....................................++
.....++
e is 65537 (0x10001)
Enter pass phrase for ca.key:
Verifying - Enter pass phrase for ca.key:
```

（2）使用 RSA 私钥创建 CA 证书。-out ca.pem 是 CA 证书的文件名。-key ca.key 是 RSA 私钥的文件名，生成 CA 证书时，使用该文件中的私钥加密。-x509 是生成自签名 CA 证书。生成自签名 CA 证书时，需要通过-days 365 设置证书的有效期，单位为天，默认情况下为 30 天。

在 Enter pass phrase for ca.key:中，输入上一步中设置的 RSA 私钥密码。

在 Country Name (2 letter code) [AU]:中，输入国家名缩写，中国为 CN。

在 State or Province Name (full name) [Some-State]:中输入公司所在省的名称，如 Sichuan。

在 Locality Name (eg, city) []:中输入公司所在城市的名称，如 Chengdu。

在 Organization Name (eg, company) [Internet Widgits Pty Ltd]:中输入公司名，如 Ghostcloud Co.,Ltd。

在 Organizational Unit Name (eg, section) []:中输入部门名，如 Development。

在 Common Name (e.g. server FQDN or YOUR name) []:中输入公司域名，如 www.ghostcloud.cn。

在 Email Address []:中输入管理员邮箱地址，如 admin@ghostcloud.cn。

```
# openssl req -new -x509 -days 365 -key ca.key -sha256 -out ca.pem
Enter pass phrase for ca.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:Sichuan
Locality Name (eg, city) []:Chengdu
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Ghostcloud Co.,Ltd
Organizational Unit Name (eg, section) []:Development
Common Name (e.g. server FQDN or YOUR name) []:www.ghostcloud.cn
Email Address []:admin@ghostcloud.cn
```

（3）创建服务器私钥和 CSR（certificate signing request）。使用 RSA 私钥生成服务器 CSR。其中，server.key 是服务器公钥的文件名，server.csr 是服务器 CSR 文件名。Common Name 为 Docker Daemon 所以在主机的主机名，如 gcserver01。

```
# openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus
.......................................... ++
.......................................................... ++
e is 65537 (0x10001)
$ openssl req -subj "/CN= gcserver01" -sha256 -new -key server.key -out
server.csr
```

（4）使用 CA 证书创建服务器证书文件。TLS 连接时，需要限制客户端的 IP 或者域名列表。在创建证书时，要添加允许的 IP 或者域名列表，只有列表中的客户端才能访问服务器。在本例中，只允许 127.0.0.1 和 192.168.1.100 的客户端访问 Docker Daemon。在测试时，可以允许所有客户端访问 Docker Daemon，此时，需要添加 0.0.0.0 的 IP。

```
# echo subjectAltName = IP:127.0.0.1,IP:192.168.1.100 > allowips.cnf
# openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca.key\
  -CAcreateserial -out server.pem -extfile allowips.cnf
Signature ok
subject=/CN=your.host.com
Getting CA Private Key
Enter pass phrase for ca.key:
# rm allowips.cnf
```

（5）创建客户端私钥和 CSR。Common Name 为 Docker Client 所在主机的主机名，如 gcclient01。

```
# openssl genrsa -out client.key 4096
Generating RSA private key, 4096 bit long modulus
```

```
.....................................................++
.............++
e is 65537 (0x10001)
# openssl req -subj '/CN= gcclient01' -new -key client.key -out client.csr
```

（6）使用 CA 证书创建客户端证书文件。需要加入 extendedKeyUsage 选项。

```
# echo extendedKeyUsage = clientAuth > client.cnf
# openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca.key\
  -CAcreateserial -out client.pem -extfile client.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca.key.pem:
# rm client.cnf
```

（7）创建服务器证书文件 server.pem 和客户端证书文件 client.pem 后，可以删除 CSR 文件。

```
#rm -v client.csr server.csr
```

（8）为了保证公钥文件的安全，需要修改公钥文件的访问权限。

```
# chmod -v 0400 ca.key server.key client.key
```

（9）为了防止篡改，需要删除证书文件的写权限。

```
# chmod -v 0444 ca.pem server.pem client.pem
```

（10）重启 Docker Daemon，加入 CA 证书、服务器证书和服务器公钥。-H=0.0.0.0:2376 表示 Docker Daemon 在 2376 端口监听。

```
# docker daemon --tlsverify --tlscacert=ca.pem --tlscert=server.pem
--tlskey=server.key \
  -H=0.0.0.0:2376
```

（11）在客户端中，运行 docker 命令时，加入 CA 证书，客户端证书和客户端公钥。本例中，只有 127.0.0.1 和 192.168.1.100 的客户端可以连接到 gcserver01 的 Docker Daemon。使用-H 连接到远端服务器的 Docker Daemon。

```
# docker --tlsverify --tlscacert=ca.pem --tlscert= client.pem --tlskey= client.key \
-H=tcp://gcserver01:2376 version
```

（12）在客户端中，可以设置环境变量，默认使用 TLS 与 gcserver01 通信。把 ca.pem、client.pem 和 client.key 复制到~/.docker 目录下。同时，设置 DOCKER HOST 和 DOCKER TLS VERIFY 环境变量。

```
# mkdir -p ~/.docker
# cp ca.pem ~/.docker/ca.pem
# cp client.pem ~/.docker/cert.pem
# cp client.key ~/.docker/key.pem
# export DOCKER HOST=tcp://$HOST:2376 DOCKER TLS VERIFY=1
```

（13）执行 docker 命令行时，默认使用 TLS 与 gcserver01 通信。

```
# docker ps
```

9.4　日志相关

以下讲解 Docker Daemon 中和日志相关的配置项。

9.4.1　-D，--debug 选项

开启调试模式，可以记录 Docker Daemon 运行时的调试信息。通过在 Docker Daemon 启动项中加入-D，或者设置日志等级为“debug”的方式，开启调试模式。当 Docker Daemon 运行在后台时，日志记录在文件中。需要收集堆栈信息时，可以发送信号量。Docker Daemon 收到 SIGUSR1 信号量，会立即把当前的堆栈信息打印到日志文件中。处理完 SIGUSER1 信号量后，Docker Daemon 会继续运行。

可以通过下面的选项配置 Docker Daemon 启动调试模式。

```
-D, --debug
```

在 Linux 系统中，通常使用 kill 发送信号量。

```
# kill -USR1 <docker daemon pid>
```

9.4.2　--log-level 选项

Docker Daemon 一般以守护进程的方式在后台运行，查看 Docker Daemon 的运行状况，只有通过日志方式。

可以通过下面的选项配置 Docker Daemon 的日志等级、日志格式等日志相关信息。

```
-l, --log-level="info"
--log-driver="json-file"
--log-opt=[]
```

不同 Linux 发行版本，日志文件的路径和查看方式都不一样，具体见表 9.3。

表 9.3　Docker Daemon 日志位置

Linux 发行版本	日志文件路径和查看方式
Ubuntu	/var/log/upstart/docker.log
Debian GNU/Linux	/var/log/daemon.log
Cent OS	less /var/log/messages| grep docker journalctl -u docker.service
Fedora	journalctl -u docker.service
Red Hat Enterprise Linux Server	Less /var/log/messages | grep docker
OpenSuSE	journalctl -u docker.service
Boot2Docker	/var/log/docker.log

-l, --log-level 设置 Docker Daemon 的日志等级，默认为 info。其他等级分别是 debug、info、warn、error、fatal。

9.4.3　--log-driver 和--log-opt 选项

--log-driver 设置容器日志驱动，默认为 json-file。其他方式分别是 json-file、syslog、journald、gelf、fluentd、awslogs、none。如果要使用 docker logs 命令查看容器日志，--log-driver 必须设置为 json-file。--log-opt 设置每一种记录方式的参数。记录方式具体见表 9.4。

表 9.4　Docker 日志记录方式

记 录 方 式	描　　述
json-file	Docker 记录容器日志的默认方式，以 json 的格式记录在文件中
none	Docker 不记录容器日志，docker logs 命令不能查看容器日志
syslog	Docker 使用 syslog 记录容器日志，日志放在 syslog 中
journald	Docker 使用 journald 记录容器日志，日志放在 journald 中
gelf	Docker 使用 Graylog Extended Log Format 记录容器日志，日志放在 GELF 服务器中
fluentd	Docker 使用 fluentd 记录容器日志，日志放在 fluentd 中
awslogs	Docker 使用 Amazon CloudWatch Logs 记录容器日志，日志放在 Amazon 服务器中
splunk	Docker 使用 splunk 记录容器日志，日志放在 splunk 服务器中
etwlogs	在 Windows 中，Docker 使用 ETW 记录容器日志，日志作为 ETW 时间存储
gcplogs	Docker 使用 Google Cloud Logging 记录容器日志，日志放在 google 服务器上

1. json-file 的参数

```
--log-opt max-size=[0-9+][k|m|g]
--log-opt max-file=[0-9+]
--log-opt labels=label1,label2
--log-opt env=env1,env2
```

max-size 设置容器日志的大小，超过这个大小的日志会回滚。如果不需要回滚容器日志，可以不设置这个参数。

max-file 设置回滚日志的最大文件数，超过这个数量的日志文件会被丢弃。如果没有设置 max-size，这个参数不生效。

同时设置 max-size 和 max-file 后，docker logs 命令值返回最新日志文件中的日志。

2. syslog 的参数

```
--log-opt syslog-address=[tcp|udp|tcp+tls]://host:port
--log-opt syslog-address=unix://path
--log-opt syslog-facility=daemon
--log-opt syslog-tls-ca-cert=/etc/ca-certificates/custom/ca.pem
--log-opt syslog-tls-cert=/etc/ca-certificates/custom/cert.pem
--log-opt syslog-tls-key=/etc/ca-certificates/custom/key.pem
--log-opt syslog-tls-skip-verify=true
--log-opt tag="mailer"
--log-opt syslog-format=[rfc5424|rfc3164]
```

默认情况下，Docker 使用容器 ID 的前 12 个字母区别容器。

syslog-address 设置 syslog 日志服务器的地址和端口。默认情况下，使用本地的

UNIX Socket 方式。使用 TCP 或者 UDP 时，默认端口为 514。

syslog-facility 设置日志类型。默认情况下，使用 daemon。其他可用的值为 kern、user、mail、daemon、auth、syslog、lpr、news、uucp、cron、authpriv、ftp、local0、local1、local2、local3、local4、local5、local6、local7。

syslog-tls-ca-cert 设置 CA 证书的绝对路径。syslog-tls-cert 设置 TLS 证书的绝对路径。syslog-tls-key 设置 TLS 公钥的绝路径。syslog-tls-skip-verify 设置为 true 时跳过 TLS 认证。syslog-address 中协议不是 TCP+TLS 时，这些参数被忽略。

syslog-format 设置 syslog 日志格式。可以是 rfc5424 或者 rfc3164。

3. gelf 参数

```
--log-opt gelf-address=udp://host:port
--log-opt tag="database"
--log-opt labels=label1,label2
--log-opt env=env1,env2
--log-opt gelf-compression-type=gzip
--log-opt gelf-compression-level=1
```

默认情况下，Docker 使用容器 ID 的前 12 个字母区别容器。

gelf-address 设置 GELF 服务器地址和端口，目前只支持 UDP 模式。

gelf-compression-type 设置日志文件的压缩方式，可以为 gzip、zlib 或者 nonw，默认为 gzip。gelf-compression-level 可以设置 gzip 和 zlib 的压缩等级，可以设置 1 到 9，默认为 1，值越大压缩率越高，压缩速度越慢。

4. fluentd 参数

fluentd-address 设置日志服务器的地址和端口。

tag 设置日志标签。

fluentd-buffer-limit 设置日志最大缓存，默认为 8MB。

fluentd-retry-wait 设置连接重试等待时间，默认为 1000ms。

fluentd-max-retries 设置连接重试次数，默认为 1073741824。

fluentd-async-connect 启动异步连接方式，默认为 false。

5. awslogs 参数

```
--log-opt awslogs-region=<aws_region>
--log-opt awslogs-group=<log_group_name>
--log-opt awslogs-stream=<log_stream_name>
```

6. splunk 参数

```
--log-opt splunk-token=<splunk_http_event_collector_token>
--log-opt splunk-url=https://your_splunk_instance:8088
```

7. gcplogs 参数

```
--log-opt gcp-project=<gcp_projext>
--log-opt labels=<label1>,<label2>
--log-opt env=<envvar1>,<envvar2>
--log-opt log-cmd=true
```

9.5 存储相关配置

9.5.1 -g, --graph 选项

设置 Docker 运行时的根目录。Docker 镜像和容器都会保存在该目录下。

通过下面的选项配置 Docker Daemon 保存镜像和容器的根目录。

```
-g, --graph="/var/lib/docker"
```

/var/lig/docker/ repositories 记录了下载到宿主机的镜像列表。使用 AUFS 存储驱动时，镜像数据保存在/var/lib/docker/aufs 中。使用 Devicemapper 存储驱动时，镜像数据保存在/var/lib/docker/devicemapper 中。容器数据保存在/var/lib/docker/container 中。

9.5.2 --storage-driver 选项

Docker 在启动容器的时候，需要创建文件系统，为 rootfs 提供挂载点。Docker Daemon 支持多种文件系统，如 AUFS、Devicemapper、Btrfs、ZFS 和 OverlayFS。不同的文件系统提供了不同的访问速度、可扩展性和附加功能。

通过下面的选项配置 Docker Daemon 的存储驱动。

```
-s, --storage-driver=""
```

1. AUFS

AUFS 是 Docker Daemon 最早支持的文件系统。最初 Docker 只能在支持 AUFS 文件系统的 Linux 发行版上运行，但是由于 AUFS 未能加入 Linux 内核，为了寻求兼容性、扩展性，Docker 在内部通过 graphdriver 机制这种可扩展的方式来实现对不同文件系统的支持。

虽然 AUFS 可能会引起内核崩溃，但是它是唯一支持在容器之间共享程序和库的文件系统。当需要同时运行上千个容器，这些容器运行运行相同的程序和库时，可以使用 AUFS。

2. Devicemapper

Devicemapper 使用了 thin-provisioning 和 copy on write 快照技术。Devicemapper 创建一个 thin pool 存储镜像和容器，这个 thin pool 创建在两个块设备上，一个块设备用于保存数据，另一个块设备用于保存元数据。默认情况下，安装 Docker 时，这两个

块设备是使用 sparse 文件模拟的。在生产环境中，可以通过 lvm，使用磁盘创建 thin pool。

3. Btrfs

Btrfs 使用 BTree 管理元数据，对文件的增加、删除、查找都非常快。使用 docker build 制作镜像时，可以使用 Btrfs，这种方式可以加快制作镜像的速度。与 Devicemapper 一样，这种方式不能在不同容器之间共享内存。

4. ZFS

ZFS 速度不如 Btrfs 快，但是是最稳定的文件系统。因为使用了 Single Copy ARC，克隆的数据启动了重复数据删除。

5. Overlay

Overlay 是一种非常快的 Union FS，已经加入 Linux 内核 3.18.0 版本中。

9.5.3 --storage-opt 选项

不同的镜像存储方式有不同的参数，可以通过--storage-opt 设置这些参数。

```
--storage-opt=[]
```

设置 Devicemapper 相关参数时，需要在参数名前加 dm 前缀；设置 ZFS 相关参数时，需要在参数名前加 zfs 前缀。

1. dm.thinpooldev

设置 thin pool 使用的块设备。使用 Devicemapper 存储驱动时，Docker Daemon 使用 thin pool 作为后端存储设备。Docker Daemon 会使用 dm.thinpooldev 中指定的块设备存储镜像和容器。最好使用专门的卷管理工具创建管理卷，这样可以使用卷管理的高级功能。在 Linux 中，一般使用 lvm 管理卷，lvm 提供手动扩容、自动扩容、动态修改 thin pool 属性等功能。

如果不提供块设备，创建 thin pool 时会创建 loopback 文件替代块设备，存储镜像和容器。使用 loopback 文件方式不需要创建卷，但是存储速度很慢，而且也不支持扩容等功能。在生产环境中，一定不要使用 loopback 文件方式。

设置 dm.thinpooldev 的方式如下。

```
# docker daemon \
--storage-opt dm.thinpooldev=/dev/mapper/thin-pool
# docker info
Storage Driver: devicemapper
 Pool Name: thin-pool
```

2. dm.basesize

设置每个镜像和容器占用的存储空间，默认值为 10GB。分配存储空间时，thin pool 采用的是按需分配的方式。初始时，只分配一个很小的空间，只有当镜像和容器需要

存储数据时，才在块设备上分配空间。

设置 dm.basesize 的方式如下。

```
# docker daemon --storage-opt dm.basesize=20G
# docker info
Storage Driver: devicemapper
 Pool Name: thin-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 21.47 GB
```

本例中，把镜像和容器的大小从 10GB 增加到 20GB，一旦镜像或容器的存储空间超过 20GB，Docker Daemon 就会报错。可以用这个参数增加镜像和容器的存储空间，但是不能减少存储空间。修改 dm.baseszie 的值后，需要重启 Docker Daemon，才能生效。

通常只在初始安装 Docker 后，修改该值。需要使用一下步骤。

```
# sudo service docker stop
# sudo rm -rf /var/lib/docker
# sudo service docker start
```

3．dm.loopdatasize

在 loopback 文件方式下，设置 data 设备使用的 loopback 文件大小，默认为 100GB。该文件是空洞文件，初始时，不会占用 100GB 空间。

设置 dm.loopdatasize 的方式如下。

```
# docker daemon --storage-opt dm.loopdatasize=100G
# docker info
Storage Driver: devicemapper
Pool Name: docker-202:1-1051822-pool
  Data file: /dev/loop0
  Data Space Used: 1.392 GB
  Data Space Total: 107.4 GB
  Data Space Available: 38.76 GB
  Data loop file: /var/lib/docker/devicemapper/devicemapper/data
  WARNING: Usage of loopback devices is strongly discouraged for production use.
Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no_warn_
on_loop_devices=true` to suppress this warning.
```

4．dm.loopmetadatasize

在 loopback 文件方式下，设置 metadata 设备使用的 loopback 文件大小，默认为 2GB。该文件也是稀疏文件，初始时，不用占用 2GB 空间。

设置 dm.loopmetadatasize 的方式如下。

```
# docker daemon --storage-opt dm.loopmetadatasize=2G
# docker info
Storage Driver: devicemapper
  Pool Name: docker-202:1-1051822-pool
```

```
 Metadata file: /dev/loop1
 Metadata Space Used: 2.699 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.145 GB
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 WARNING: Usage of loopback devices is strongly discouraged for production use.
 Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no_warn_on_
 loop_devices=true` to suppress this warning.
```

5. dm.fs

设置 thin-pool 中使用的文件系统，默认为 xfs，支持 ext4 和 xfs。

设置 dm.fs 的方式如下。

```
# docker daemon --storage-opt dm.fs=ext4
# docker info
Storage Driver: devicemapper
Pool Name: thin-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 21.47 GB
 Backing Filesystem: ext4
```

6. dm.mkfsarg

设置创建文件系统时使用的参数。

设置 dm.mkfsarg 的方式如下。

```
# docker daemon --storage-opt "dm.mkfsarg=-O ^has_journal"
```

dm.mountopt

设置挂载文件系统时，使用的参数。

设置 dm.mountopt 的方式如下。

```
# docker daemon --storage-opt dm.mountopt=nodiscard
```

dm.blocksize

设置创建 thin pool 时，使用的 block size，默认为 64KB。

设置 dm.blocksize 的方式如下。

```
# docker daemon --storage-opt dm.blocksize=64K
# docker info
Storage Driver: devicemapper
 Pool Name: thin-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 21.47 GB
```

7. dm.blkdiscard

设置是否开启释放模式，默认开启。thin pool 使用按需分配的方式分配存储空间，在 Linux kernel 3.8 之前的版本中，当执行删除容器、删除镜像、删除容器内文件时，

不会释放这些空间。Docker 增加了 dm.blkdicard 参数，作为变通方案。开启后，Docker 会释放存储空间。

有人会担心，关闭释放模式后，删除容器内文件时，不释放存储空间，会造成容器空间不足。例如，限制容器使用 20GB 空间，创建了一个 10GB 文件并删除，再创建一个 10GB 文件时，Docker 会不会提示空间不足？实际上不会出现这种情况，当空间增加到容器空间上限时，系统会重新利用之前已经删除的文件占用空间。这就像处理脏数据一样，一开始并不回收脏数据，当空间不足时，系统会清理脏数据，释放空间。

在 Linux kernel 3.8 之后的版本中，可以关闭释放模式，使用内核中 dm-thin 提供的释放存储空间功能。

设置 dm.blkdiscard 的方式如下。

```
# docker daemon --storage-opt dm.blkdiscard=false
```

8. dm.override_udev_sync_check

设置是否覆盖 udev 同步检查。

可以使用 docker info 查看宿主机是否支持 udev 同步。

```
# docker info
Storage Driver: devicemapper
Udev Sync Supported: true
```

当宿主机支持 udev 同步时，Devicemapper 存储驱动会和 udev 协商管理容器使用的设备。当宿主机不支持 udev 同步时，Devicemapper 存储驱动会和 udev 竞争管理设备。这种竞争会引起错误。

在不支持 udev 同步的宿主机上，Docker Daemon 使用 Devicemapper 存储驱动会失败，必须开启 dm.override_udev_sync_check。发生竞争关系时，Docker Daemon 会打印错误，并继续运行。

```
# docker daemon --storage-opt dm.override_udev_sync_check=true
```

9. dm.use_deferred_removal

设置该参数为 true，在支持异步删除设备的宿主机上，开启异步删除。

一些版本的 libdem 和 Linux 内核支持异步删除设备机制。在删除设备时，如果设备被其他程序使用，可以暂时不删除，把删除操作加到设备的操作列表中。当设备空闲时，再删除。

通常容器退出时，Docker Daemon 会删除它所使用的临时存储空间。如果这些空间不能被立即删除，容器退出会处于死循环。开启 dm.use_deferred_removal 后，容器立即退出，系统会在空间空闲时，删除它们。

设置 dm.use_deferred_removal 的方式如下。

```
# docker daemon --storage-opt dm.use_deferred_removal=true
# docker info
Storage Driver: devicemapper
  Deferred Removal Enabled: true
```

10. dm.use_deferred_deletion

设置该参数为 true，开启异步删除 thin pool 设备。

删除容器前，Docker Daemon 会删除容器使用的 thin pool 设备。默认情况下，删除这些设备是同步的。如果删除 thin pool 设备失败，删除容器也会失败，Docker Daemon 会和报错。

```
Error deleting container: Error response from daemon: Cannot destroy container
```

开启 dm.use_deferred_removal 和 dm.use_deferred_delete 可以避免这种错误。此时，Devicemapper 存储驱动会把 thin pool 设备标记为已删除，然后删除容器。当 thin pool 设备空闲时，再异步删除它们。

在生产环境中，可以开启这两个选项。

设置 dm.use_deferred_deletion 的方式如下。

```
# docker daemon \
    --storage-opt dm.use_deferred_deletion=true \
    --storage-opt dm.use_deferred_removal=true
# docker info
Storage Driver: devicemapper
  Deferred Removal Enabled: true
  Deferred Deletion Enabled: true
```

11. dm.min_free_space

设置可用空间阀值，单位为百分比，默认值为 10%。

当为镜像和容器分配存储空间时，Devicemapper 存储驱动会检查 thin pool 中的可用空间。如果可用空间少于最小可用空间，Docker 相关的操作会失败。dm.min_free_space 设置了最小可用空间的百分比，从 0%到 99%。设置 0%，关闭可用空间检查。检查可用空间时，会同时检查 data 可用空间和 metadata 可用空间。

运维人员需要监控 thin pool 中的可用空间，当空间不够时，可以删除镜像和容器，或者为 thin pool 添加磁盘。

设置 dm.min_free_space 的方式如下。

```
# docker daemon --storage-opt dm.min_free_space=10%
```

12. zfs.fsname

设置 ZFS 的数据集名字。

设置 zfs.fsname 的方式如下。

```
# docker daemon -s zfs --storage-opt zfs.fs
```

9.6　网桥相关配置

Docker 安装完成后，会建立一个名叫 docker0 的虚拟网桥，用于容器与外界通信。默认情况下，Docker 自动配置 docker0 的 IP、子网掩码和容器的 IP 范围。用户也可以自己配置这些值。

9.6.1　--bip 选项

通过下面的选项配置 docker0 的 IP 和子网掩码。

```
--bip=CIDR
```

下例中，配置 docker0 的 IP 和子网掩码为 172.0.0.1/16。

```
# docker daemon --bip=172.0.0.1/24
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.0.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        ether 02:42:9b:c5:09:ea  txqueuelen 0  (Ethernet)
        RX packets 2280766  bytes 385011475 (367.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2935851  bytes 355880285 (339.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

9.6.2　--fixed-cidr，--fixed-cidr-v6 选项

通过下面的选项配置容器 IP 的范围。

```
--fixed-cidr=CIDR, --fixed-cidr-v6=CIDR
```

创建新容器时，Docker Daemon 会从--fixed-cidr 配置的 IP 范围中，选择一个可用的 IP 分配给容器的 eth0 网卡，并把 docker0 的子网掩码配置给 eth0。--fixed-cidr-v6 设置 IPv6 的 IP 范围。

```
# docker daemon --bip=172.0.0.1/16 --fixed-cidr=172.0.1.1/24
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
```

docker0 的 IP 为 172.0.0.1，子网掩码为 255.255.0.0。

```
# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
```

```
        inet 172.0.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        ether 02:42:9b:c5:09:ea  txqueuelen 0  (Ethernet)
        RX packets 2280797  bytes 385013563 (367.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2935851  bytes 355880285 (339.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

创建的容器 IP 为 172.0.1.0 网段中的 IP。

docker create --rm -it ubuntu bash

```
root@b64f220f5a08:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:00:01:00
          inet addr:172.0.1.0  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe00:100/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

9.6.3　--mtu 选项

通过下面的选项配置 docker0 的最大传输单元长度。

```
--mtu=BYTES
```

MTU 越大，网络传输效率越高而传输延迟增大，所以要根据网络传输效率和传输延迟选择合适的 MTU。

9.6.4　-b, --bridge 选项

默认情况下，Docker Daemon 使用 docker0 作为网桥。用户可以创建自己的网桥，然后指定 Docker Daemon 使用该网桥。

通过下面的选项配置网桥信息。

```
--bridge=CIDR
```

如果 Docker Daemon 已经运行，则需要停止 Docker Daemon，删除 docker0。

```
# service docker stop
# ip link set dev docker0 down
# brctl delbr docker0
# iptables -t nat -F POSTROUTING
```

手动创建新网桥 bridge0。

```
# brctl addbr bridge0
# ip addr add 192.168.5.1/24 dev bridge0
# ip link set dev bridge0 up
```

查看网桥已经启动。

```
# ip addr show bridge0
```

```
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
```

启动 Docker Daemon，加入新创建的网桥。

```
# docker daemon --bridge=bridge0
```

确定 iptables 中已经添加了 NAT masquerade 记录。

```
# iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.5.0/24      0.0.0.0/0
```

此时，创建容器，会把容器的 eth0 绑定到网桥 bridge0 上。容器的 IP 范围为网桥 bridge0 中的 IP 范围。

9.7　容器与外部通信

容器是否能够与外部通信，需要在宿主机上配置两个参数。

（1）设置系统的 ip_forward 为 1，允许宿主机想外转发。

```
# cat /proc/sys/net/ipv4/ip_forward
1
```

（2）设置 iptables 规则，允许对外连接。

可以在 Docker Daemon 中设置对应参数，让 Docker Daemon 完成以上配置。

9.7.1　--ip-forward 选项

设置--ip-forward=true，Docker Daemon 启动时，会自动修改宿主机的 ip_forward 为 1。默认值为 true。

9.7.2　--iptables 选项

设置--iptables=true，Docker Daemon 启动时，会在 iptables 中追加转发规则。默认值为 true。

9.7.3　--ip, --ipv6 选项

启动容器时，如果不设置导出端口，外部主机就不能访问容器。设置导出端口需要在 docker run 命令中加入-P 或者-p 选项，此时，Docker Daemon 会为容器分配一个

主机 IP 和端口。-P 把镜像中指定的所有暴露端口导出，导出端口随机分配，从 32768 到 61000。-p 为容器指定特定 IP 和导出端口，格式为-p IP:host_port:container_port。当容器有多个 IP 时，可以绑定导出端口到其中的一个 IP 上，此时，可以在 Docker Daemon 中设置--ip=IP_ADDR。--ipv6 设置 IPv6 地址。

```
--ip=IP_ADDR, --ipv6=IPv6_ADDR
```

例如，宿主机有两个 IP，一个为公网 IP，另一个为私网 IP。

```
# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
       inet 192.168.0.1  netmask 255.255.240.0  broadcast 0.0.0.0
       ……

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
       inet 10.35.21.8  netmask 255.255.252.0  broadcast 10.24.207.255
       ……

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
       inet 120.135.21.15  netmask 255.255.252.0  broadcast 120.25.95.255
       ……
```

为了对外提供服务，把容器的导出端口绑定到公网 IP 上。

```
# docker daemon --ip=120.135.21.35
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
```

导出端口时，容器自动绑定到公网 IP 上。

```
# docker run --rm -it -p 80 httpd
# docker ps
CONTAINER ID        IMAGE                                  COMMAND
CREATED             STATUS                  PORTS                       NAMES
249ba7ad642a        httpd  "httpd-foreground"   1 seconds ago      Up Less than a
second   120.135.21.35:32769->80/tcp   sharp_bohr
```

9.8　其他网络配置

9.8.1　--default-gateway、--default-gateway-v6 选项

默认情况下，容器使用 docker0 的 IP 作为网关。也可以通过--default-gateway 设置新的容器网关。通过下面的选项配置 IPv6 的网关。

```
--default-gateway="", --default-gateway-v6=""
```

9.8.2 --dns，--dns-opt，--dns-search 选项

通过下面的选项配置 DNS 的地址。

```
--dns=[], --dns-opt=[], --dns-search=[]
```

通过下面的选项配置默认的搜索域。

```
# docker daemon --dns 8.8.8.8
```

通过下面的选项配置 DNS 选项。

```
# docker daemon --dns-search ghsotcloud.cn
```

9.9 execdriver 配置

在 Docker 中，对容器管理的模块为 execdriver。目前，Docker 支持的容器管理方式为 native，使用 libcontainer 进行容器管理，包括 Namespaces 的管理、CGroups 的管理、rootfs 的启动配置、Linux capability 权限集，以及进程运行的环境变量配置等，具体如图 9.1 所示。

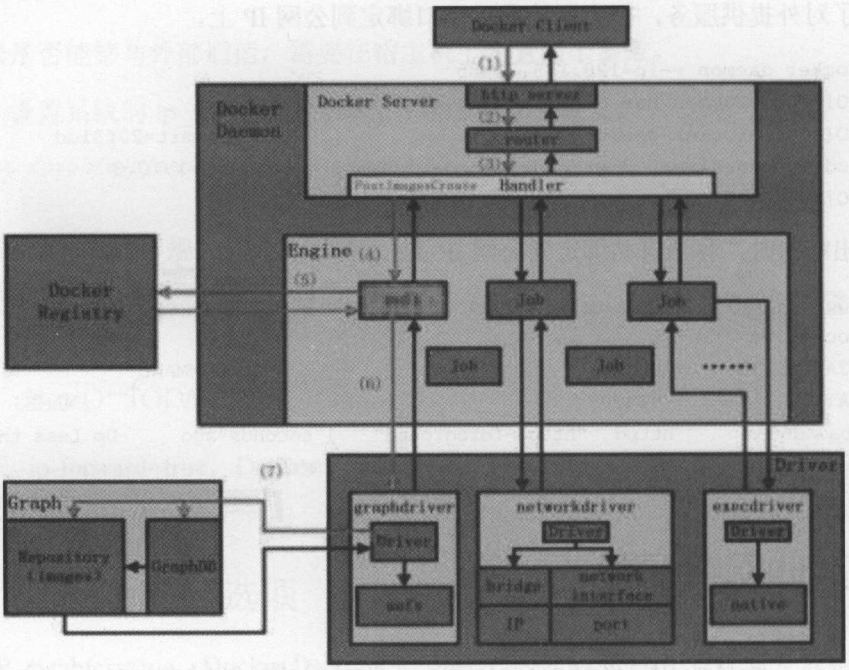


图 9.1　Docker 中的 execdriver

9.9.1 --exec-opt 选项

可以通过--exec-opt 设置 execdriver 参数。使用 libcontainer 时，所有参数都以 native 作为前缀，如 native.cgroupdriver。该参数设置如何管理容器的 CGroups，默认值为 cgroupfs，可选值为 cgroupfs 和 systemd。

配置--exec-opt 的方式如下。

```
# docker daemon --exec-opt native.cgroupdriver=systemd
```

在 Windows 的容器中，使用--exec-opt 配置其他参数。例如，Windows 中，默认的容器隔离技术是 process，可以修改为 hyperv。

```
# docker daemon --exec-opt isolation=hyperv
```

9.9.2　--exec-root 选项

设置 execdriver 使用的状态文件的根目录，默认为/var/run/docker。

```
# ls /var/run/docker
execdriver  netns
```

9.10　其他配置

为容器设置默认的 ulimit 参数。启动容器时，可以添加--ulimit，为容器设置单独的 ulimit 参数。若启动容器时没有添加—ulimit 参数，将使用 Docker Daemon 中--default-ulimit 提供的参数。若启动容器时添加了--ulimit 参数，将覆盖 Docker Daemon 中--default-ulimit 提供的参数。

使用--default-ulimit 和--ulimit 设置 nproc 参数的值时，需要特别注意。在 Linux 中，nproc 是设置一个用户能够使用的最大进程数，而不是一个容器使用的最大进程数。

```
# docker daemon --default-ulimit nofile=1024:1024
```

9.11　习题

本章详细介绍了 Docker Daemon 的各种选项。接下来，通过习题和实验检验本章的学习成果。

（1）修改 Docker Daemon 启动项文件，添加国内镜像加速仓库，--registry-mirror=http://mirror.ghostcloud.cn。

（2）修改 Docker Daemon 启动项文件，开启 debug 模式。并修改日志登记为 warn。

（3）找一台有两个 IP 的主机，修改 Docker Daemon 启动项文件，修改 Docker Daemon 的监听 IP 和端口，-H tcp://ip1:2375。通过远端主机，访问 ip1 的 Docker Daemon。

第 10 章　如何编写 Dockerfile

镜像为 Docker 提供了丰富多彩的应用，可以说，正是因为有各种镜像，才使 Docker 流行起来。制作镜像需要编写 Dockerfile。如何缩减镜像体积，暴露哪些端口，设置哪些挂载卷都是由 Dockerfile 决定的。写好 Dockerfile，就能够让编译出来的镜像简洁而优雅，Dockerfile 指令见表 10.1。

表 10.1　Dockerfie 指令列表

指　令	说　明
FROM	设置镜像使用的基础镜像
MAINTAINER	设置镜像的作者
RUN	编译镜像时，运行的脚本
CMD	设置容器的启动命令
LABEL	设置镜像的标签
EXPOSE	设置镜像的暴露端口
ENV	设置容器的环境变量
ADD	编译镜像时，复制文件到镜像中
COPY	编译镜像时，复制文件到镜像中
ENTRYPOINT	设置容器的入口程序
VOLUME	设置容器的挂载卷
USER	设置执行 RUN、CMD 和 ENTRYPOINT 的用户名
WORKDIR	设置 RUN、CMD、ENTRYPOINT、COPY 和 ADD 指令的工作目录
ARG	设置编译镜像时，加入的参数
ONBUILD	设置镜像的 ONBUILD 指令
STOPSIGNAL	设置容器的退出信号量

10.1　本地编译镜像

Dockerfile 类似 Makefile，是用类似脚本的语言编写的。用户使用 docker build 命令编译镜像。

docker build 命令可以设置编译镜像时，使用的 CPU 数量、内存大小、Dockerfile 文件路径、编译参数等。docker build 命令的用法如下。

```
# docker build --help

Usage:  docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

  --build-arg=[]                  Set build-time variables
  --cpu-shares                    CPU shares (relative weight)
  --cgroup-parent                 Optional parent cgroup for the container
  --cpu-period                    Limit the CPU CFS (Completely Fair Scheduler) period
  --cpu-quota                     Limit the CPU CFS (Completely Fair Scheduler) quota
  --cpuset-cpus                   CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems                   MEMs in which to allow execution (0-3, 0,1)
  --disable-content-trust=true    Skip image verification
  -f, --file                      Name of the Dockerfile (Default is 'PATH/Dockerfile')
  --force-rm                      Always remove intermediate containers
  --help                          Print usage
  --isolation                     Container isolation level
  -m, --memory                    Memory limit
  --memory-swap                   Swap limit equal to memory plus swap: '-1' to enable
                                  unlimited swap
  --no-cache                      Do not use cache when building the image
  --pull                          Always attempt to pull a newer version of the image
  -q, --quiet                     Suppress the build output and print image ID on success
  --rm=true                       Remove intermediate containers after a successful build
  --shm-size                      Size of /dev/shm, default value is 64MB
  -t, --tag=[]                    Name and optionally a tag in the 'name:tag' format
  --ulimit=[]                     Ulimit options
```

编译镜像由 Docker Daemon 完成。这是最简单的编译过程。

```
# docker build -t <image_name> .
Sending build context to Docker daemon  6.51 MB
```

docker build 的命令格式。

```
Usage:  docker build [OPTIONS] PATH | URL | -
```

PATH 是编译镜像使用的工作目录，Docker Daemon 在编译开始时，扫描 PATH 中的所有文件。应该尽量在一个干净的目录中编译镜像，如果编译目录中文件太多，扫描文件会花费很长时间。切忌使用系统根目录/作为编译工作目录，否则 Docker Daemon 会扫描文件系统上的所有文件。可以在编译目录中加入.dockerignore，过滤不需要的文件。Docker Daemon 在编译时，会忽略.dockerignore 中的文件。

默认情况下，Docker Daemon 会从 PATH 或 GIT 中读取 Dockerfile。如果要使用其他 Dockerfile，可以使用-f 参数。

```
# docker build -f ~/php.Dockerfile .
```

使用-t 给镜像打标签。

```
# docker build -t ghostcloud.cn/myapp .
```

可以给镜像打不同标签。

```
# docker build -t ghostcloud.cn/myapp:2.0 -t ghostcloud.cn/myapp:latest .
```

Docker Daemon 从 Dockerfile 每次顺序读取一条指令，生成一个临时容器，在容器中执行指令。编译成功，把容器提交为一个中间镜像，作为镜像层，加入最终镜像。

Docker Daemon 采用缓存机制，加快编译过程。如果在缓存中找到需要的中间镜像，直接把使用该镜像，不生成临时容器。加入--no-cache，在编译镜像时，不使用缓存。

下面是编译镜像的过程。

```
# docker build -t svendowideit/ambassador .
Sending build context to Docker daemon 15.36 kB
Step 0 : FROM alpine:3.2
 ---> 31f630c65071
Step 1 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 2a1c91448f5f
Step 2 : RUN apk update &&      apk add socat &&        rm -r /var/cache/
 ---> Using cache
 ---> 21ed6e7fbb73
Step 3 : CMD env | grep _TCP= | (sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):
\(.*\)/socat -t 100000000 TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' && echo
wait) | sh
 ---> Using cache
 ---> 7ea8aef582cc
Successfully built 7ea8aef582cc
```

10.2　dockerignore 文件

编译开始之前，Docker Daemon 会读取编译目录中的.dockerignore 文件，忽略其中的文件和目录，这和 git 很像。可以在.dockerignore 文件中加入文件和目录规则，忽略多个文件。

```
*/temp*
*/*/temp*
temp?
```

*/temp*忽略一级子目录下，以 temp 开头的所有文件和目录。如 PATH/somedir/temporary.txt 和 PATH/somddir/temp。

//temp* 忽略二级子目录下，以 temp 开头的所有文件和目录。如

PATH/somedir/subdir/temp.txt 和 PATH/somedir/subdir/temp。

temp? 忽略当前目录下，以 temp 开头，名字长度为 5 个字符的文件和目录。如 PATH/tempa 和 PATH/tempb。

```
*.md
!README.md
```

Docker Daemon 会读取!后面的文件。上例中，Docker Daemon 会读取 README.md，并忽略所有 md 文件。

10.3　Dockerfile 格式

Dockerfile 中每条指令由指令+参数组成，中间以空格隔离。#后面为注释内容，#可以写在任何位置。

```
# Comment
INSTRUCTION arguments
```

指令不区分大小写。在实践中，为了便于区分，一般指令使用大写，参数使用小写。Dockerfile 中的第一条指令必须是 FROM，设置基础镜像。

```
FROM busybox
RUN echo 'we are running some # of cool things'
```

10.4　Dockerfile 指令详解

Dockerfile 是 Docker 在编译镜像时执行的脚本文件，由很多条指令组成，这些指令按顺序排列。每条指令在编译镜像的过程中，执行相应的程序，完成某项功能。Dockerfile 类似 Makefile，有自己的格式和支持的指令。Docker 在编译镜像时，会解决指令间的依赖关系，并按顺序执行下去。下面将介绍每条指令的功能。

10.4.1　FROM 指令

Dockerfile 的第一条指令是 FROM 指令。镜像都是从一个基础镜像生成的。基础镜像可以是一个操作系统镜像，也可以是其他镜像。在基础镜像中，加入提供服务的程序和程序需要的依赖包，定制自己的镜像。

下面是 FROM 指令的格式。

```
FROM <image>
FROM <image>:<tag>
FROM <image>@<digest>
```

Docker Hub 上提供了常见的 Linux 操作系统的镜像，如 Ubuntu，CentOS，Debian，

Fedora 等。操作系统的基础镜像通常有几百 MB，目前，最小的操作系统镜像为 Alpine Linux，只有 5MB。使用 Alpine Linux 作为基础镜像，可以缩写镜像体积。

可以在一个 Dockerfile 中加入多条 FROM 指令，一次生成多个镜像。两条 FROM 指令之间的内容放在一个镜像中。

tag 和 digest 是可选项。如果忽略 tag，会使用 latest 镜像。

10.4.2　MAINTAINER 指令

使用 MAINTAINER 指令设置镜像作者。

下面是 MAINTAINER 指令的格式。

```
MAINTAINER <name>
```

10.4.3　RUN 指令

RUN 指令会生成一个新的容器，在容器中执行脚本，这个容器使用当前的镜像。脚本正常执行完后，Docker Daemon 会把该容器提交为一个中间镜像，供后面的指令使用。

下面是 RUN 指令的格式。

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

RUN 指令有两种方式。第一种方式为 shell 方式，使用/bin/sh -c <command>运行脚本。可以在这种方式中使用\，把脚本分成多行。

```
RUN source $HOME/.bashrc ;\
echo $HOME
```

第二种方式为 exec 方式。镜像中没有/bin/sh，或者要使用其他 shell，使用这种方式。

```
RUN ["/bin/bash", "-c", "echo hello"]
```

注意　exec 方式使用 JSON 数组格式，必须使用双引号"，而不是单引号'。

与 shell 不同，这种方式不会调用 shell 命令。与在终端中执行命令有一些区别。例如，RUN ["echo", "$HOME"]，不会读取环境变量 HOME。如果要调用 shell，则必须使用 RUN ["sh", "-c", "echo", "$HOME"]。

10.4.4　CMD 指令

使用 CMD 指令设置容器的启动命令，本书随后将介绍 CMD、ENTRYPOINT 和 RUN 的区别。Dockerfile 中只能有一条 CMD 指令，如果写了多条 CMD 指令，只有最

后一条 CMD 指令生效。

下面是 CMD 指令的格式。

```
CMD ["executable","param1","param2"]
CMD ["param1","param2"]
CMD command param1 param2
```

CMD 有三种运行方式。第一种是 exec 方式，第二种是 ENTRYPOINT 参数方式，第三种是 shell 方式。第一种与第三种的区别可以参考 RUN 指令。第二种方式，为 entrypoint 提供参数列表。

注意　exec 方式使用 JSON 数组格式，必须使用双引号"，而不是单引号'。

与 shell 不同，这种方式不会调用 shell 命令。与在终端中执行命令有一些区别。例如，CMD["echo", "$HOME"]，不会读取环境变量 HOME。如果要调用 shell，必须使用 CMD["sh", "-c", "echo", "$HOME"]。

10.4.5　LABEL 指令

使用 CMD 指令设置镜像的标签，可以通过 docker inspect 查看标签。

下面是 CMD 指令的格式。

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

每个标签采用 Key=Value 的格式，不同标签之前通过空格隔离。可以使用"和\。

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

每条指令都会生成一个镜像层，Docker 规定一个镜像最多只能有 127 层。如果超过 127 层，Docker Daemon 会报错。

```
Error response from daemon: Cannot create container with more than 127 parents
```

应该尽量减少镜像层的数量。需要设置多个标签时，最好使用一条 LABEL 指令。

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

使用连接符\，可以强加阅读性。

```
LABEL multi.label1="value1" \
     multi.label2="value2" \
other="value3"
```

镜像会继承基础镜像的标签，LABEL 指令会覆盖基础镜像中的同名标签。

通过 docker inspect 可以查看镜像标签。

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple
     lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
}
```

10.4.6　EXPOSE 指令

使用 EXPOSE 指令设置镜像暴露端口，记录容器启动时监听在哪些端口。

下面是 EXPOSE 指令的格式。

```
EXPOSE <port> [<port>...]
```

当容器启动时，Docker Daemon 会扫描镜像中的暴露端口。如果加入-P 参数，Docker Daemon 会把镜像中的所有暴露端口全部导出，并为每个暴露端口分配一个随机的主机端口。暴露端口是容器的监听端口；主机的导出端口是外部访问容器的端口，两个可以不一样。例如，把 httpd 的 80 端口导出到 8080 端口，访问 Web Server 时，需要加上 8080 端口。

```
# docker run -d -p 8080:80 httpd
# curl <hostip>:8080
```

EXPOSE 指令只是设置暴露端口，并不导出端口。启动容器时，需要使用-P 或者-p 参数设置导出端口，此时，才能通过外部访问容器提供的服务。

通过 docker inspect 可以查看镜像的暴露端口。

```
"Config": {
          "Hostname": "e5c68db50333",
          "Domainname": "",
          "User": "",
          "AttachStdin": false,
          "AttachStdout": false,
          "AttachStderr": false,
          "ExposedPorts": {
              "80/tcp": {}
          },
}
```

10.4.7　ENV 指令

使用 ENV 指令设置镜像中的环境变量。

下面是 ENV 指令的格式。

```
ENV <key> <value>
ENV <key>=<value> ...
```

在 Dockerfile 中，可以设置环境变量。这些变量在整个编译周期都有效。变量名一般大写，便于区分。

第一种方式，一次设置一个环境变量。

```
ENV NAME ghostcloud
ENV COUNTRY China
ENV CITY Cheng du
```

第二种方式，一次设置多个环境变量。最佳实践建议使用第二种方式，这样可以减少中间镜像数量，加快编译时间。

```
ENV NAME=ghostcloud COUNTRY=China CITY="Cheng du"
```

通过$variable 或者${variable }使用变量。两种方式效果相同，不过${variable}更严格，如${foo}_bar。

```
ENV HTTPD_VERSION=httpd-2.2.31
RUN wget -qO /${HTTPD_TAR_GZ} /tmp
```

此外，${variable}还支持一些高级功能。

- ${variable:-word} 如果没有定义 variable，则使用 word 作为默认值。
- ${variable:+word} 如果定义了 variable，则使用 word 覆盖 variable 中的值。否则，使用空字符串。

使用\取消读取变量值，如\$foo 和\${foo}分配输出字符串$foo 和${foo}。

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}   # WORKDIR /bar
ADD . $foo       # ADD . /bar
COPY \$foo /quux # COPY $foo /quux
```

以下指令支持读取环境变量。其中，Docker 1.4 以后的版本，才支持 ONBUILD。

- ADD
- COPY
- ENV
- EXPOSE
- LABEL
- USER
- WORKDIR
- VOLUME

- STOPSIGNAL
- ONBUILD

注意　环境变量可能会引起副作用。例如，在 Debian 的基础镜像中，加入 ENV DEBIAN_FRONTEND noninteractive，会覆盖 apt-get 中的使用环境变量。此时，最好在一条指令中定义环境变量 RUN <key>=<value> <command>。

10.4.8　ADD 指令

使用 ADD 指令，把文件复制到镜像中。

下面是 ADD 指令的格式。

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

当路径中有空格时，需要使用第二种方式。

把文件复制到镜像中，src 可以是文件、目录或则 URL，支持通配符。当 src 是文件或目录时，Docker Daemon 会从编译目录寻找这些文件或目录。

```
ADD hom* /mydir/        # 从编译目录中，复制所有前缀是 hom 的文件。
ADD hom?.txt /mydir/    # 替换？为任一字符，如 home.txt
```

dest 是镜像中的绝对路径，或者相对于 WORKDIR 的路径。

```
ADD test relativeDir/       # 复制编译目录中的 test 目录到'WORKDIR'/relativeDir/中。
ADD test /absoluteDir/      # 复制编译目录中的 test 目录到/absoluteDir 中。
```

在镜像中，目标文件和目录的 UID 和 GID 都是 0。

如果 src 为 URL，复制到镜像后，文件或目录的权限是 600。如果 HTTP 中包含 Last-Modified，文件的 mtime 将使用这个值。

注意　如果 URL 使用了认证机制，不能使用 ADD 指令。需要使用 RUN wget 或者 RUN curl。

如果源文件变化，ADD 指令会取消编译缓存功能。ADD 以后的指令都不能使用编译缓存。

ADD 指令遵循以下规则。

- src 必须在编译目录中。不能使用 ADD ../something /something。
- src 为 URL 时，如果 dest 结尾没有/，则 dest 作为文件名。从 URL 下载文件，存储在 dest 路径。

```
ADD http://example.com/file /tmp/file
```

- http://example.com/file 下载 file，存储在/tmp/file。
- src 为 URL 时，如果 dest 结尾有/，则 dest 作为存储目录。从 URL 下载文件，保存在 dest 目录中。

```
ADD http://example.com/file /tmp/folder/
```

- 从 http://example.com/file 下载文件，保存在/tmp/folder/file。
- 如果 src 为目录，复制目录内所有内容，包括文件系统的元数据。不复制目录本身。
- 如果 src 是编译目录中的压缩文件，压缩方式为 identity, gzip, bzip2 or xz。ADD 指令会把它解压成目录。
- 如果 src 是一个文件，则复制文件和元数据。
- 如果 src 使用了通配符，或者 src 是一个文件列表，则 dest 必须以/结尾。
- 如果 dest 不以/结尾，则 dest 为文件名。
- 如果 dest 不存在，则 ADD 会自动创建 dest 及缺失的上级目录，类似# mkdir -p /folder/folder/file。

10.4.9　COPY 指令

也可以使用 COPY 指令，把文件复制到镜像中。

下面是 COPY 指令的格式。

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"]
```

当路径中有空格时，需要使用第二种方式。

把文件或者目录复制到容器中，src 可以是文件、目录，支持通配符。Docker Daemon 会从编译目录寻找这些文件或目录。

```
COPY hom* /mydir/          # 从编译目录中，复制所有前缀是 hom 的文件。
COPY hom?.txt /mydir/      # 替换？为任意字符，如 home.txt
```

dest 是镜像中的绝对路径，或者相对于 WORKDIR 的路径。

```
COPY test relativeDir/     # 复制编译目录中的 test 目录到'WORKDIR'/relativeDir/中。
COPY test /absoluteDir/    # 复制编译目录中的 test 目录到/absoluteDir 中。
```

在镜像中，目标文件和目录的 UID 和 GID 都是 0。

COPY 遵循以下规则。

- src 必须在编译目录中。不能使用 COPY ../something /something。
- 如果 src 为目录，则复制目录内所有内容，包括文件系统的元数据。不复制目

录本身。

- 如果 src 是一个文件，则复制文件和元数据。
- 如果 src 使用了通配符，或者 src 是一个文件列表，则 dest 必须以/结尾。
- 如果 dest 不以/结尾，则 dest 为文件名。
- 如果 dest 不存在，则 ADD 会自动创建 dest 及缺失的上级目录，类似# mkdir -p /folder/folder/file。

10.4.10　ENTRYPOINT 指令

使用 ENTRYPOINT 指令设置容器的入口程序。

下面是 ENTRYPOINT 指令的格式。

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

入口程序有两种格式：第一种为 exec 方式，第二种为 shell 方式。第二种方式使用 /bin/sh -c 运行入口程序，此时入口程序不能接收信号量。运行 docker stop <container> 时，入口程序不能接收 SIGTERM。

入口程序是容器启动时执行的程序，docker run 命令中最后的命令将作为参数传递给入口程序。例如，docker run <image> -d，将把-d 传给入口程序。

如果 Dockerfile 中有多条 ENTRYPOINT，只有最后的 ENTRYPOINT 指令生效。

1. 使用 exec 方式

第一种为 exec 方式。

```
ENTRYPOINT ["executable", "param1", "param2"]
```

在 Dockerfile 中，配置入口程序，容器启动时，以 exec 方式运行入口程序。

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

启动容器，使用 top 命令查看容器中的进程信息，入口程序的 PID 为 1。

```
# docker run -it --rm --name test top -H
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R   0.0  0.1   0:00.04 top
```

通过 docker exec 连接到容器，查看进程。

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps aux
```

在 Dockerfile 中，使用 ENTRYPOINT 启动 httpd 服务器。

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

如果使用脚本作为入口程序，则需要保证脚本的最后一个程序要能够接收信号量。可以在脚本最后使用 exec 或者 gosu 启动传入脚本的命令。

```
#!/bin/bash
set -e

# 判断第一个参数是否是 postgres
if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    # 如果 PGDATA 目录为空，则执行初始化。
    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

# 启动传入脚本的命令
exec "$@"
```

如果在容器退出时需要做一些清理工作，则需要保证脚本能够接收信号量，传递信号量给清理程序。

```
#!/bin/sh
# Note: I've written this using sh so it works in the busybox container too

# 容器停止时，使用 trap 捕捉信号量
trap "echo TRAPed signal" HUP INT QUIT TERM

# 启动 Apache 服务
/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# 停止服务，并做清理工作
echo "stopping apache"
```

```
/usr/sbin/apachectl stop

echo "exited $0"
```

启动容器后，可以通过 docker exec 或者 docker top 监控容器进程。

```
# docker run -it --rm -p 80:80 --name test apache
# docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0   4448   692 ?        Ss+  00:42   0:00 /bin/sh /run.sh 123 cmd cmd2
root        19  0.0  0.2  71304  4440 ?        Ss   00:42   0:00 /usr/sbin/apache2 -k start
www-data    20  0.2  0.2 360468  6004 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
www-data    21  0.2  0.2 360468  6000 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
root        81  0.0  0.1  15572  2140 ?        R+   00:44   0:00 ps aux
# docker top test
PID                 USER                COMMAND
10035               root                {run.sh} /bin/sh /run.sh 123 cmd cmd2
10054               root                /usr/sbin/apache2 -k start
10055               33                  /usr/sbin/apache2 -k start
10056               33                  /usr/sbin/apache2 -k start
# /usr/bin/time docker stop test
test
real 0m 0.27s
user 0m 0.03s
sys  0m 0.03s
```

注意　启动容器时，加入--entrypoint=bin，可以替换镜像中 ENTRYPOINT 设置的入口程序。此时，入口程序以 exec 的方式运行。

exec 方式使用 JSON 数组格式，必须使用双引号"，而不是单引号'。

与 shell 不同，这种方式不会调用 shell 命令。与在终端中执行命令有一些区别。例如，ENTRYPOINT["echo", "$HOME"]，不会读取环境变量 HOME。如果要调用 shell，必须使用 ENTRYPOINT["sh", "-c", "echo", "$HOME"]。

2. 使用 shell 方式

第二种为 shell 方式。容器启动时，通过/bin/sh -c 运行入口程序。此时，将忽略 CMD 指令和 docker run 中的参数。

```
ENTRYPOINT command param1 param2
```

为了保证容器能够接收 docker stop 发送的信号量，需要通过 exec 启动程序。

```
FROM ubuntu
ENTRYPOINT exec top -b
```

启动容器，使用 top 命令查看容器中的进程信息。容器中只有一个进程，PID 为 1。

```
# docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
```

```
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

通过 docker stop 命令，容器可以正常退出。

```
# /usr/bin/time docker stop test
test
real    0m 0.20s
user    0m 0.02s
sys 0m 0.04s
```

如果在 ENTRYPOINT 指令中没有加入 exec 命令，则启动容器时容器中会出现两个进程。

```
FROM ubuntu
ENTRYPOINT top -b
CMD --ignored-param1
```

启动容器时，若使用 top 命令查看容器中的进程信息，则发现有两个进程，PID 为 1 的进程是 shell 程序。

```
# docker run -it --name test top --ignored-param2
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K cached
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% sirq
Load average: 0.01 0.02 0.05 2/101 7
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     S     3168   0%   0% /bin/sh -c top -b cmd cmd2
    7     1 root     R     3164   0%   0% top -b
```

通过 docker stop 命令，容器不能正常退出，容器不能接收 docker stop 发送的 SIGTREM 信号。超时以后，docker stop 会发送 SIGNKILL，强制停止容器。可以发现超时时间大概是 10s，real 0m 10.19s。

```
# docker exec -it test ps aux
PID   USER     COMMAND
    1 root     /bin/sh -c top -b cmd cmd2
    7 root     top -b
    8 root     ps aux
$ /usr/bin/time docker stop test
test
real    0m 10.19s
user    0m 0.04s
sys 0m 0.03s
```

10.4.11　VOLUME 指令

使用 VOLUME 指令设置容器的挂载点。

下面是 VOLUME 指令的格式。

```
VOLUME ["/data"]
VOLUME /data1 /data2
```

Docker Daemon 会把主机目录或者数据卷容器挂载到这些挂载点。第一种格式为 JSON 数据，必须使用双引号。第二种格式把多个目录通过空格连接起来。

启动容器时，Docker Daemon 会新建挂载点，并用镜像中的数据初始化挂载点。

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

从这个镜像启动容器时，会新建/myvol 目录，并把 greeting 文件复制到/myvol 中。

注意　如果 VOLUME /mydata 之后的指令修改了/mydata 目录中的内容，编译镜像时，这些修改会被忽略。

10.4.12　USER 指令

使用 USER 指令设置执行 RUN、CMD 和 ENTRYPOINT 的用户名或 UID。

下面是 USER 指令的格式。

```
USER daemon
```

10.4.13　WORKDIR 指令

使用 WORKDIR 指令设置 RUN、CMD、ENTRYPOINT、ADD 和 COPY 指令的工作目录。

下面是 WORKDIR 指令的格式。

```
WORKDIR /path/to/workdir
```

如果工作目录不存在，则 Docker Daemon 会自动创建。

可以在 Dockerfile 中不同地方调用 WORKDIR 指令。如果 WORKDIR 跟的是相对路径，这个路径会跟在上一条 WORKDIR 指定的路径后面。下例中，pwd 为/a/b/c。

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

WORKDIR 可以使用 ENV 指令中定义的环境变量。下例中，pwd 为/path/$DIRNAME。

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

10.4.14　ARG 指令

使用 ARG 指令设置编译变量。

下面是 ARG 指令的格式。

```
ARG <name>[=<default value>]
```

编译镜像时，可以通过 docker build --build-arg <var>=<value>设置这些变量。如果 docker build 设置的编译变量，没有通过 ARG 定义，则 Docker Daemon 会报错。

```
One or more build-args were not consumed, failing build.
```

可以定义多个编译变量。

```
FROM busybox
ARG user1
ARG buildno
...
```

可以为编译变量设置默认值。

```
FROM busybox
ARG user1=someuser
ARG buildno=1
...
```

ARG 从定义它的地方开始生效，而不是调用的地方。

```
1 FROM busybox
2 USER ${user:-some_user}
3 ARG user
4 USER $user
```

编译时，设置编译变量 user=what_user。

```
# docker build --build-arg user=what_user Dockerfile
```

第 2 行，没有定义编译变量 user，user 为空，USER 使用默认值 some_user。

第 3 行，定义编译变量 user。user 开始生效，后面的指令可以使用 user。

第 4 行，设置编译变量 user。使用 docker build 传入的值 what_user。

在 ARG 指令之前，调用编译变量时，编译变量的值总为空。

注意　不要使用编译变量设置敏感信息，如 github 密钥和用户的认证信息。

可以使用 ENV 或 ARG 设置 RUN 使用的变量。如果 ENV 和 ARG 定义了同名变量，ENV 定义的值会覆盖 ARG 定义的值。

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER v1.0.0
```

```
4 RUN echo $CONT_IMG_VER
```

编译时，设置编译变量 CONT_IMG_VER=v2.0.1。

```
# docker build --build-arg CONT_IMG_VER=v2.0.1 Dockerfile
```

此时，在 RUN 指令中，CONT_IMG_VER 是 v1.0.0。

可以灵活地使用 ENV 和 ARG。与 ARG 不同，ENV 设置的变量值在整个编译过程中，总是保持不变的。

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER ${CONT_IMG_VER:-v1.0.0}
4 RUN echo $CONT_IMG_VER
```

编译时，不设置编译变量。

```
# docker build Dockerfile
```

在第 3 行中，设置 CONT_IMG_VER 的默认值为 v1.0.0。在编译过程中，CONT_IMG_VER 的值一直是 v1.0.0，并且保持不变。

Dockerfile 中有一些预定义的 ARG 编译变量。例如：

- HTTP_PROXY
- http_proxy
- HTTPS_PROXY
- https_proxy
- FTP_PROXY
- ftp_proxy
- NO_PROXY
- no_proxy

ARG 的变量值在编译过程中是可变的，这会对使用编译缓存造成影响。

这里有两个 Dockerfile。

第 1 个 Dockerfile 如下：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo $CONT_IMG_VER
```

第 2 个 Dockerfile 如下：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo hello
```

编译时，使用--build-arg CONT_IMG_VER=<value>设置 CONT_IMG_VER 的值。设置的 CONT_IMG_VER 值不同，编译过程也不同。

第 2 行是完全相同，编译时，两个 Dockerfile 可以使用相同的中间镜像。

当 CONT_IMG_VER=hello 时，第 3 行完全相通，编译时，两个 Dockerfile 可以使用相同的中间镜像。当 CONT_IMG_VER!=hello 时，第 3 行指令不同，编译时，两个 Dockerfile 不能使用同一个中间镜像。

10.4.15　ONBUILD 指令

使用 ONBUILD 指令设置子镜像的编译钩子指令。

下面是 ONBUILD 指令的格式。

```
ONBUILD [INSTRUCTION]
```

当从该镜像生成子镜像时，会调用 ONBUILD 指令。在子镜像的编译过程中，首先会执行父镜像中的 ONBUILD 指令。所有的编译指令都可以成为钩子指令。

制作基础镜像时，可以使用该指令，如制作编译容器。制作通用的 Python 程序编译容器时，需要把 Python 源代码复制到容器中特定的位置，某些情况下，还需要调用特定的编译脚本。不同的 Python 程序源码放在不同的位置中，因此不能使用通用的 ADD 和 RUN 指令。一种可选的做法是使用 Dockerfile 模板，针对不同的 Python 源代码，在模板上做修改。这种方法不易更新，容器出错，而且效率非常低。

此时，应该使用 ONBUILD 指令进行优化。ONBUILD 的流程如下。

（1）编译时，读取所有 ONBUILD 指令，并记录下来。在当前的编译过程中，不执行这些的指令。

（2）生成镜像时，把所有 ONBUILD 指令记录在镜像的配置文件的 OnBuild 关键字中。可以通过 docker inspect 查看 ONBUILD 指令。

（3）使用 FROM 指令，从基础镜像生成新的镜像。执行 FROM 指令时，会读取基础镜像的 ONBUILD 指令，并顺序执行。如果执行 ONBUILD 指令失败，则编译中断。当所有 ONBUILD 指令执行成功后，开始执行子镜像中的指令。

（4）子镜像不会继承基础镜像的 ONBUILD 指令。生成子镜像时，ONBUILD 指令不会记录在子镜像中。

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

10.4.16　STOPSIGNAL 指令

使用 STOPSIGNAL 指令设置容器退出时，Docker Daemon 向容器发送的信号量。

下面是 STOPSIGNAL 指令的格式。

```
STOPSIGNAL signal
```

信号量可以是数字或者信号量名字。如，9 或者 SIGKILL。

10.5　CMD、ENTRYPOINT 和 RUN 的区别

RUN 指令是设置编译镜像时执行的脚本和程序，镜像编译完成，RUN 指令的生命周期结束。

容器启动时，可以通过 CMD 和 ENTRYPOINT 设置启动程序，两者有很大的区别和紧密的联系。

CMD 叫作容器默认启动命令，既然是默认，就可以被替换掉。若在 docker run 命令末尾添加 Command，则替换镜像中 CMD 设置的启动程序。

```
FROM ubuntu
CMD ["echo", "I amd default CMD."]
```

启动容器时，若在 docker run 末尾不添加 Command，运行镜像中 CMD 设置的启动程序，此时，启动容器，打印"I amd default CMD."。

```
# docker run --rm test
I am default CMD.
```

启动容器时，在 docker run 末尾添加 echo "I am from docker run."，这条命令会替换镜像中的 CMD。此时，启动容器，打印"I am from docker run."。

```
# docker run --rm test echo "I am from docker run."
I am from docker run.
```

ENTRYPOINT 叫做入口程序，不能被 docker run 命令末尾的 Command 替换。docker run 命令末尾的 Command 会被当作字符串，传递给 ENTRYPOINT，作为参数。

```
FROM ubuntu
ENTRYPOINT ["ps"]
```

启动容器时，在 docker run 末尾不添加 Command，运行镜像中 ENTRYPOINT 设置的启动程序，此时，启动容器，打印 ps 的输出。

```
# docker run --rm test
  PID TTY          TIME CMD
1 ?        00:00:00 ps
```

启动容器时，在 docker run 末尾添加-ef，-ef 作为参数传递给 ENTRYPOINT。此

时，启动容器，打印 ps -ef 的输出。

```
# docker run --rm test -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 03:56 ?        00:00:00 ps -ef
```

可以在 docker run 中，加入--entrypoint 替换镜像中的入口程序。

```
# docker run --rm --entrypoint hostname test-ent
588cf5dc9f15
```

以下是使用 CMD 和 ENTRYPOINT 指令时，需要遵循的一些规则。

- 在 Dockerfile 中，应至少有一条 CMD 或 ENTRYPOINT 指令。
- 当使用容器作为一个程序容器时，应使用 ENTRYPOINT 定义入口程序。例如，swarm 容器就是一个程序行容器，通过容器提供 swarm 服务。

```
docker run --rm swarm manage --heplp
Usage: swarm manage [OPTIONS] <discovery>
Manage a docker cluster
Arguments:
  <discovery>    discovery service to use [$SWARM_DISCOVERY]
                 * token://<token>
                 * consul://<ip>/<path>
                 * etcd://<ip1>,<ip2>/<path>
                 * file://path/to/file
                 * zk://<ip1>,<ip2>/<path>
                 * [nodes://]<ip1>,<ip2>
```

- 在 Dockerfile 中，如果同时定义了 ENTRYPOINT 和 CMD，CMD 会作为参数传递给 ENTYRPOINT。

```
FROM ubuntu
ENTRYPOINT ["ps"]
CMD ["-ef"]
# docker run --rm test
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 03:56 ?        00:00:00 ps -ef
```

- 在 docker run 末尾加入 Command，会覆盖镜像中的 CMD。
- 在 docker run 中加入--entrypoint，会覆盖镜像中的 ENTRYPOINT。

10.6　习题

本章详细介绍了如何通过 Dockerfile 制作镜像。接下来，通过习题和实验检验本章的学习成果。

（1）使用 Ubuntu:14.04 镜像作为基础镜像，制定自己的镜像。使用 MAINTAINER 指令，设置制作者为 ghostcloud。使用 LABEL 指令，添加 version=1.0，company=

ghostcloud。使用 docker build 命令编译镜像，并通过 docker inspect 查看镜像信息。

（2）使用 Ubuntu:14.04 镜像作为基础镜像，制作自己的镜像。使用 EXPOSE 指令导出 22 端口，使用 VOLUME 指令设置容器挂在点为/tmp。使用 docker build 命令编译镜像，并通过 docker inspect 查看镜像信息。

（3）使用 Ubuntu:14.04 镜像作为基础镜像，制作自己的镜像。在主机上，镜像编译目录中新建 README.txt 文件，在文件中添加"This is ghostcloud customized image."。使用 COPY 指令，把该文件复制到容器的/目录下。使用 docker build 命令编译镜像，从该镜像启动容器，进入容器，检查容器中/README.txt 文件内容。

第 11 章　Dockerfile 最佳实践

通过 Dockerfile 可以制作镜像，通过优化 Dockerfile 中的指令，可以减少镜像体积。按照一些规范制作 Dockerfile，可以增加 Dockerfile 的可读性和可维护性。下面将介绍制作 Dockerfile 的最佳实践。

11.1　基本原则

1. 容器的生命期是短暂的

通过 Dockerfile 制作出镜像，从镜像启动容器，这些容器的生命期应尽量短暂。短暂意味着容器随时可以被停止和删除。通过简单的配置，立即可以重新启动一个新的容器。

2. 使用.dockerignore

最好在空目录中编译镜像，编译目录中只包含制作镜像所需的文件。

docker build 编译镜像时，会扫描编译目录中的文件和文件夹。编译目录是一个本地目录或一个 GIT 代码库。

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
...
```

编译镜像由 Docker Daemon 完成，一开始，Docker Daemon 会读取编译目录中的所有文件和文件夹。为了加快镜像的编译过程，应该把 Dockerfile 放在一个干净的编译目录中，只把容器需要的文件放在编译目录中。切忌使用系统根目录（/），否则编译镜像时，会扫描整个文件系统。

3. 只安装需要的包

为了减少镜像体积和编译时间，应避免安装额外的、不需要的包。例如，不要在一个数据库的镜像中安装编辑器。

4．每个容器中只运行一个进程

一般情况下，在一个容器中只运行一个进程。设置应用程序时，应该尽量分解为不同模块，每个模块跑在一个容器中。这样可以方便地水平扩展容器，也方便重用容器。如果一个进程依赖另一个进程，则可以使用 link 的方式，把这两个容器联系起来。

5．减少镜像层

Dockerfile 中的每条指令都会生成一个镜像层，一个镜像最多只有 127 个镜像层。在编辑 Dockerfile 时，需要评估可读性和指令的数量。

6．把多个参数排在不同的行中

当一条指令中有多个参数时，使用字母方式排序。这样可以提高指令的可读性，避免用户重复写入参数，方便以后添加和更新参数。使用连接符（\）连接多行。

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

7．编译缓存

编译镜像时，Docker Daemon 从 Dockerfile 中读出所有指令，按顺序一条一条地执行。每执行一条指令，都会生成一个镜像层。Docker Daemon 会在镜像缓存中检查镜像层是否存在，如果不存在，则创建一个新的镜像层。如果不想使用镜像缓存中的镜像层，则可以在编译镜像时加上--no-cache=true。

Docker Daemon 使用镜像缓存时，遵循了一些基本规则，开发者在编写 Dockerfile 时需要了解 Docker Daemon 使用镜像缓存的机制。下面是 Docker Daemon 使用镜像缓存的一些基本原则。

- Docker Daemon 从基础镜像编译出新的镜像。首先，Docker Daemon 会读取 FROM 后面的第一条指令。然后，Docker Daemon 在镜像缓存中寻找所有从相同基础镜像生成的子镜像。接下来，Docker Daemon 在所有的子镜像中，检查是否有子镜像使用了相同的指令。如果找到，就使用该子镜像作为镜像层。否则，生成新的镜像层。

  一般情况下，Docker Daemon 通过指令在镜像缓存中寻找镜像层。但是，某些情况下，Docker Daemon 还会检查其他信息。

- 针对 ADD 和 COPY 指令，Docker Daemon 会检查镜像层中所有源文件的元数据和文件内容。在检查元数据时，不会检查最后修改时间和最后访问时间。Docker Daemon 会寻找相应的缓存镜像层，并检查镜像层中对应文件的元数据和文件内容是否与当前文件一致。如果一致，就使用该镜像层。否则，生成新的镜像层。

- 除了 ADD 和 COPY 指令外，Docker Daemon 在镜像缓存中寻找镜像层时，不会检查文件。例如，在执行 RUN apt-get -y update 时，Docker Daemon 不会检查镜像层中的文件变化，仅比较 RUN 指令本身。

11.2　Dockerfile 指令最佳实践

Dockerfile 中的每条指令都有相应的优化方法和最佳实践。这些技巧能够帮助开发者减小镜像体积，方便维护 Dockerfile 更新，解决编译时间。下面将详细介绍每一条指令的优化技巧。

11.2.1　FROM 指令最佳实践

尽量使用官方镜像作为基础镜像。推荐使用 Debian，因为该镜像体积很小，只有 150MB 左右，而且 Docker 对它进行了优化。

11.2.2　RUN 指令最佳实践

使用 RUN 指令时，应该保证 RUN 中执行的命令有很好的可读性，容易理解，方便维护。使用\，把长的 RUN 指令分成多行。

在 RUN 指令中，最常用的是 apt-get 命令。开发者使用 RUN apt-get 安装依赖包和程序，需要使用一些技巧，优化 apt-get 命令。

避免使用 RUN apt-get upgrade 和 dist-upgrade 更新基础镜像中的所有包和程序。在没有使用-privileged 的运行容器中，基础镜像中的很多基础包是不能升级的。如果基础镜像中的某个包过期了，开发者可以联系这个包的维护者。如果开发者需要升级某个特定的包，如 foo，可以使用 apt-get install -y foo 自动升级。

必须在一条 RUN 指令中同时执行 apt-get update 和 apt-get install。

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

因为存在镜像缓存机制，所以如果在两条 RUN 指令中执行 apt-get update 和 apt-get install，则可能会造成 apt-get install 失败。

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl
```

编译完成该镜像后，镜像缓存中会保存所有的镜像层。假设，开发者编写了另外一个镜像的 Dockerfile，其中使用了几条相同的指令。Docker Daemon 编译新镜像时，

在镜像缓存中发现 RUN apt-get update 镜像层，Docker Daemon 会直接使用该镜像层。然后执行接下来的指令 RUN apt-get update。此时，Docker Daemon 并没有运行 RUN apt-get update 指令，安装包时，包数据并没有更新，这将造成用户编译新镜像时，始终不能安装新的 curl 和 Nginx。

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

使用 RUN apt-get update && apt-get install -y，保证每次编译镜像时，都是安装的最新的包。这种技术叫作“缓存失效”（cache busting）。安装包时，也可以指定版本号，忽略 apt 缓存中的包信息，执行精确安装。指定版本号可以避免包不兼容的问题。

```
RUN apt-get update && apt-get install -y \
   package-bar \
   package-baz \
   package-foo=1.3.*
```

尽量在一条 RUN 指令中安装需要的包。如果包很多，则可以采用字母顺序排列，这样方便维护。每行列出一个安装包，避免重复安装。

```
   RUN apt-get update && apt-get install -y \
   aufs-tools \
   automake \
   build-essential \
   curl \
   dpkg-sig \
   libcap-dev \
   libsqlite3-dev \
   mercurial \
   reprepro \
   ruby1.9.1 \
   ruby1.9.1-dev \
   s3cmd=1.1.* \
&& rm -rf /var/lib/apt/lists/*
```

本例在安装指令中，指定了 s3cmd 的版本号，如果镜像中是一个旧版本，这条指令可以强制升级到指定版本。

安装结束后，应清理安装缓存，减小镜像体积。使用 rm -rf /var/lib/apt/lists/*删除 apt-get 的缓存目录。RUN 指令开头使用了 apt-get update，保证每次 apt-get install 时都是使用的最新包信息。

Gitlab 官方镜像中使用的 RUN 指令如下。

```
FROM ubuntu:14.04
MAINTAINER Sytse Sijbrandij

# 使用一条指令安装需要的包。
# 多个包按字母顺序排列，分成多行，使用连接符连接。
RUN apt-get update -q \
```

```
    && DEBIAN_FRONTEND=noninteractive apt-get install -yq --no-install-recommends \
      ca-certificates \
      openssh-server \
      wget \
      apt-transport-https \
      vim \
      nano

# 下载 gpg 公钥，安装 gitlab-ce。
RUN echo "deb https://packages.gitlab.com/gitlab/gitlab-ce/ubuntu/ `lsb_release
-cs` main" > /etc/apt/sources.list.d/gitlab_gitlab-ce.list
RUN wget -q -O - https://packages.gitlab.com/gpg.key | apt-key add -
RUN apt-get update && apt-get install -yq --no-install-recommends gitlab-ce

# 配置 SSHD。
RUN mkdir -p /opt/gitlab/sv/sshd/supervise \
    && mkfifo /opt/gitlab/sv/sshd/supervise/ok \
    && printf "#!/bin/sh\nexec 2>&1\numask 077\nexec /usr/sbin/sshd -D" >
    /opt/gitlab/sv/sshd/run \
    && chmod a+x /opt/gitlab/sv/sshd/run \
    && ln -s /opt/gitlab/sv/sshd /opt/gitlab/service \
    && mkdir -p /var/run/sshd

# 在 SSHD 中，不使用 DNS。
RUN echo "UseDNS no" >> /etc/ssh/sshd_config

# 配置默认参数。
RUN ( \
  echo "" && \
  echo "# Docker options" && \
  echo "# Prevent Postgres from trying to allocate 25% of total memory" && \
  echo "postgresql['shared_buffers'] = '1MB'" ) >> /etc/gitlab/gitlab.rb && \
  mkdir -p /assets/ && \
  cp /etc/gitlab/gitlab.rb /assets/gitlab.rb

# 导出 SSHD 端口和服务器端口。
EXPOSE 443 80 22

# 设置导出卷。
VOLUME ["/etc/gitlab", "/var/opt/gitlab", "/var/log/gitlab"]

# 复制 assets 文件。
COPY assets/wrapper /usr/local/bin/

# 设置启动程序。
CMD ["/usr/local/bin/wrapper"]
```

11.2.3　CMD 指令最佳实践

CMD 指令设置镜像中的默认启动命令和参数。容器启动时，如果在 docker run 命

令中没有加入启动命令，执行镜像中 CMD 设置的默认启动命令。设置启动命令时，应尽量使用 JSON 格式，CMD ["executable", "param1", "param2"…]。例如，制作一个提供 Apache 服务的镜像时，可以这样设置启动命令。

```
CMD ["apache2","-DFOREGROUND"]
```

大部分情况下，可以使用脚本程序启动服务。

```
CMD ["perl", "-de0"]
CMD ["python"]
CMD [“php”, “-a”]
```

Django 官方镜像中使用的 CMD 指令。

```
FROM python:2.7

# 创建/usr/src/app 目录。
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# 设置 ONBUILD 指令。
ONBUILD COPY requirements.txt /usr/src/app/
ONBUILD RUN pip install --no-cache-dir -r requirements.txt
ONBUILD COPY . /usr/src/app

# 安装软件包。
RUN apt-get update && apt-get install -y \
    gcc \
    gettext \
    mysql-client libmysqlclient-dev \
    postgresql-client libpq-dev \
    sqlite3 \
  --no-install-recommends && rm -rf /var/lib/apt/lists/*

# 导出 8000 端口。
EXPOSE 8000

# 设置启动程序。
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

一般情况下，不要使用 CMD 设置 ENTRYPOINT 的参数，除非开发者和使用者都很熟悉 ENTRYPOINT 的工作原理。

```
ENTRYPOINT ["program"]
CMD ["param1", "param2"]
```

11.2.4　EXPOSE 指令最佳实践

EXPOSE 指令设置镜像中的导出端口。容器启动后，将在这些端口监听。通常，在镜像中导出常用端口。

在提供 Apache 服务的镜像中，导出 80 端口。

```
FROM debian:jessie

# 设置环境变量。
ENV HTTPD_PREFIX /usr/local/apache2
ENV PATH $PATH:$HTTPD_PREFIX/bin

# 新建文件夹，并把工作目录切换到新建的文件夹。
RUN mkdir -p "$HTTPD_PREFIX" \
  && chown www-data:www-data "$HTTPD_PREFIX"
WORKDIR $HTTPD_PREFIX

……
# 复制文件到镜像。
COPY httpd-foreground /usr/local/bin/

# 导出 Web 端口。
EXPOSE 80

# 设置启动程序。
CMD ["httpd-foreground"]
```

在提供 MongoDB 服务的镜像中，导出 27017 端口。

```
FROM ubuntu:14.04
MAINTAINER Tutum Labs <support@tutum.co>

# 安装软件包。
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10 && \
    echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-
org/3.2 multiverse" | tee /etc/apt/sources.list.d/mongodb-org-3.2.list && \
    apt-get update && \
    apt-get install -y --force-yes pwgen mongodb-org mongodb-org-server mongodb-
org-shell mongodb-org-mongos mongodb-org-tools && \
    echo "mongodb-org hold" | dpkg --set-selections && echo "mongodb-org-server
hold" | dpkg --set-selections && \
    echo "mongodb-org-shell hold" | dpkg --set-selections && \
    echo "mongodb-org-mongos hold" | dpkg --set-selections && \
    echo "mongodb-org-tools hold" | dpkg --set-selections

# 导出/data/db 目录。
VOLUME /data/db

# 设置环境变量。
ENV AUTH yes
ENV STORAGE_ENGINE wiredTiger
ENV JOURNALING yes

# 复制文件到镜像。
ADD run.sh /run.sh
ADD set_mongodb_password.sh /set_mongodb_password.sh
```

```
# 导出 27017 和 28017 端口。
EXPOSE 27017 28017

# 设置启动程序。
CMD ["/run.sh"]
```

镜像的使用者在启动容器时，可以通过添加参数-p 或者-P 设置导出端口的外部访问端口。链接容器时，可以使用环境变量设置端口，如 MYSQL_PORT_3306_TCP。

11.2.5　ENV 指令最佳实践

ENV 指令设置镜像中的环境变量。使用容器对外提供服务时，最好通过环境变量设置服务相关配置。通过这种方式，可以方便地在不同环境中修改配置，快速启动服务，加快开发、测试、部署流程。在环境变量中修改 PATH，可以保证 Nginx 正常启动。

```
ENV PATH /usr/local/nginx/bin:$PATH
CMD ["nginx"]
```

ENV 可以设置服务相关配置，如 Postgres 中的 PGDATA。通过环境变量设置应用的版本号，可以方便运维人员维护镜像。相对于在程序中通过常量或硬编码设置版本号，这种方式更灵活。

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/
src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

squid 官方镜像中使用的 ENV 指令。

```
FROM sameersbn/ubuntu:14.04.20160608
MAINTAINER sameer@damagehead.com

# 设置环境变量。
ENV SQUID_VERSION=3.3.8 \
    SQUID_CACHE_DIR=/var/spool/squid3 \
    SQUID_LOG_DIR=/var/log/squid3 \
    SQUID_USER=proxy

# 安装软件包。
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
80F70E11F0F0D5F10CB20E62F5DA5F09C3173AA6 \
 && echo "deb http://ppa.launchpad.net/brightbox/squid-ssl/ubuntu trusty main" >>
 /etc/apt/sources.list \
 && apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y squid3-ssl=${SQUID_VERSION}* \
 && mv /etc/squid3/squid.conf /etc/squid3/squid.conf.dist \
 && rm -rf /var/lib/apt/lists/*
```

```
# 复制配置文件和脚本到镜像。
COPY squid.conf /etc/squid3/squid.conf
COPY entrypoint.sh /sbin/entrypoint.sh

# 修改脚本权限。
RUN chmod 755 /sbin/entrypoint.sh

# 以 TCP 方式导出 3128 端口。
EXPOSE 3128/tcp

# 导出目录。
VOLUME ["${SQUID_CACHE_DIR}"]

# 设置入口程序。
ENTRYPOINT ["/sbin/entrypoint.sh"]
```

11.2.6　ADD 和 COPY 指令最佳实践

尽管 ADD 和 COPY 指令功能相似，在实际应用中，还是推荐使用 COPY 指令。因为 COPY 指令在功能上比 ADD 指令更单一。COPY 指令仅把编译目录中的文件复制到镜像中。ADD 指令除本地复制功能外，还会解压 tar 文件，并支持远端复制。ADD 指令的最佳应用场景是，把编译目录中的 tar 文件复制到镜像中，并在镜像中解压该文件。

```
ADD rootfs.tar.xz /
```

在制作镜像时，如果需要复制多个文件，最好使用多条 COPY 指令，而不要使用一条 COPY 指令复制所有文件。通过这种方式，如果某些文件发生变动，则在编译镜像时可以使用编译缓存。

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

把 COPY . /tmp/指令加载 RUN 指令之后。如果 requirements.txt 文件变动，在编译新镜像时，还可以使用 COPY . /tmp/指令的编译缓存。

如果要添加远程文件，则不要使用 ADD 指令。因为 ADD 指令解压 tar 文件以后，不会删除源文件，这会增加镜像的体积。

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

下载远程文件时，最好使用 RUN curl 指令或者 RUN wget 指令代替 ADD。完成源码编译后，得到可以运行的二进制文件。此时，不需要在镜像中保留源码，可以删除源码，以减小镜像体积。

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
```

```
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
    && rm -rf /usr/src/things
```

Alpine Linux 官方镜像中使用的 ADD 指令如下所示。

```
FROM scratch
ADD rootfs.tar.gz /
```

如果仅需要复制编译目录中的文件和目录，而不需要自解压，最好使用 COPY 指令。

一个自定义的 Nginx 镜像中使用的 COPY 指令如下所示。

```
FROM ubuntu:14.04

MAINTAINER owner

# 设置镜像的标签信息。
LABEL Name="CustomerNginx" \
    Version="1.0.0"

# 配置 apt 源信息。
RUN { \
        echo "deb http://mirrors.aliyun.com/ubuntu/ trusty main restricted
        universe multiverse"; \
        echo "deb http://mirrors.aliyun.com/ubuntu/ trusty-security main
        restricted universe multiverse"; \
  } >/etc/apt/sources.list

# 安装软件包。
RUN apt-get update && apt-get install -y \
    gcc \
    libpcre3 libpcre3-dev libssl-dev\
    make \
    openssl \
    perl \
    wget \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*

# 设置环境变量。
ENV OPENRESTY_VERSION ngx_openresty-1.9.7.1
ENV OPENREST_TAR $OPENRESTY_VERSION.tar.gz

# 下载 openresty 源文件，并编译。
RUN mkdir -p /home/openresty \
  && cd /home/openresty \
  && wget -qO /home/openresty/$OPENREST_TAR
  https://openresty.org/download/$OPENREST_TAR \
  && tar -zxf /home/openresty/$OPENREST_TAR \
  && cd /home/openresty/$OPENRESTY_VERSION \
```

```
  && perl configure --with-luajit \
  && make \
  && make install \
  && cd /home \
  && rm -rf /home/openresty \
  && mkdir -p /openresty/logs /openresty/conf

# 修改环境变量 PATH，追加/usr/local/openresty/nginx/sbin 目录。
ENV PATH /usr/local/openresty/nginx/sbin:$PATH

# 导出 Web 端口。
EXPOSE 80

# 复制 openresty 的配置文件和 lua 脚本。
COPY nginx.conf /openresty/conf/
COPY lua/ /openresty/lua/

# 修改目录和文件的属性及权限。
RUN chmod +x /openresty/lua/*.lua \
  && touch /openresty/portmap.json \
  && chown nobody /openresty/portmap.json

# 复制入口程序脚本到镜像。
COPY entrypoint.sh /entrypoint.sh

# 为入口程序脚本添加执行权限。
RUN chmod +x /entrypoint.sh

# 设置镜像的入口程序。
ENTRYPOINT ["/entrypoint.sh"];
```

11.2.7　ENTRYPOINT 指令最佳实践

当需要把容器当作一个命令行工具使用时，最好通过 ENTRYPOINT 指令设置镜像的入口程序。

在下例中，制作一个 s3cmd 的镜像，提供与 s3cmd 命令一模一样的功能。

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

启动这个容器后，可以看到 s3cmd 命令的帮助文档。

```
$ docker run s3cmd
Usage: s3cmd [options] COMMAND [parameters]

S3cmd is a tool for managing objects in Amazon S3 storage. It allows for
making and removing "buckets" and uploading, downloading and removing
"objects" from these buckets.
```

启动容器时，添加参数，可以完成相关功能。

```
$ docker run s3cmd ls s3://mybucket
```

启动容器时，如果需要提供额外的服务，则可以把这些服务写到脚本中。通过 ENTRYPOINT 指令，把教程设置为入口程序。例如，如果需要在启动 CentOS 的容器时打开 sshd 服务，则可以使用 CentOS 作为基础镜像。在模板镜像中添加启动脚本，脚本中会开启 sshd 服务。这样，每次启动容器时都会执行启动脚本，首先开启 sshd 服务，然后执行 docker run 中的服务。

Postgres 官方镜像的 Dockerfile 如下所示。

```
FROM centos:7

# 设置作者信息。
MAINTAINER owner

# 设置镜像标签信息。
LABEL  Name="centos-ssh" \
Version="1.0.0" \

# 导出 SSHD 的 22 端口。
EXPOSE 22

# 复制入口程序脚本到镜像中，并添加运行权限。
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

# 设置镜像的入口程序，并使用 CMD 设置入口程序的参数。
ENTRYPOINT ["/entrypoint.sh"]
CMD ["/bin/bash"]
```

Postgres 官方镜像使用启动脚本作为入口程序。

```
#!/bin/bash
set -e

# 判断脚本的第一个参数是否为 postgres。
if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    # 如果 PGDATA 目录不存在，则需要初始化 postgres。
    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

# 执行传入脚本的程序。
exec "$@"
```

在启动脚本中，使用 exec 命令启动服务，可以让这个服务在容器中使用 PID 1 作为进程号。此时，这个服务可以接收发送给容器的 UNIX 信号量。

在 Dockerfile 中，把 Postgres 的启动脚本复制到镜像中。

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

启动脚本为 Postgres 镜像提供了更加丰富的功能。

单纯启动 Postgres 如下所示。

```
$ docker run postgres
```

启动 Postgres，并把参数传递给服务器。

```
$ docker run postgres postgres --help
Postgres is the PostgreSQL server.

Usage:
  Postgres [OPTION]...

Options:
  -B NBUFFERS         number of shared buffers
  -c NAME=VALUE       set run-time parameter
  -C NAME             print value of run-time parameter, then exit
  -d 1-5              debugging level
  -D DATADIR          database directory
  -e                  use European date input format (DMY)
  -F                  turn fsync off
  -h HOSTNAME         host name or IP address to listen on
  -i                  enable TCP/IP connections
  -k DIRECTORY        Unix-domain socket location
  -l                  enable SSL connections
  -N MAX-CONNECT      maximum number of allowed connections
  -o OPTIONS          pass "OPTIONS" to each server process (obsolete)
  -p PORT             port number to listen on
  -s                  show statistics after each query
  -S WORK-MEM         set amount of memory for sorts (in kB)
  -V, --version       output version information, then exit
  --NAME=VALUE        set run-time parameter
  --describe-config   describe configuration parameters, then exit
  -?, --help          show this help, then exit
...
```

启动 Postgres，直接进入 bash，进行交互式操作。

```
$ docker run --rm -it postgres bash
```

11.2.8　VOLUME 指令最佳实践

VOLUME 指令设置镜像中的挂载卷。如果需要在容器中，对数据库、配置文件、用户上传文件夹等文件目录做数据持久化，则可以使用 VOLUME 指令导出这些文件和目录。使用宿主机的存储设备保存这些文件目录。

启动容器时，Docker Daemon 会从 VOLUME 中读取挂载点列表，在容器中创建这些挂载点。然后，在主机的/var/lib/docker/volumes 目录中创建对应的目录，把这些目录挂载到容器中。

```
FROM ubuntu:14.04
VOLUME ["/mydata"]
```

启动容器，Docker Daemon 会根据 VOLUME 自动创建挂载点。

```
# docker run --rm -it test bash
root@76aa84975694:/#
```

查询容器的详细信息。在 Mounts 字段中，发现在主机上创建了/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9e1dcc095f35/_data 目录，把该目录挂载到容器中的/mydata 挂载点。

```
# docker inspect 76aa84975694
"Mounts": [
            {
                "Name":
"1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9e1dcc095f35",
                "Source":
"/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9e1d
cc095f35/_data",
                "Destination": "/mydata",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ],
```

在容器中，/mydata 目录为空，主机的对应目录也没有内容。

```
# tree
/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9
e1dcc095f35/
/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9
e1dcc095f35/
└── _data
```

在容器中，在/mydata 目录下，新建文件 myfile。

```
root@76aa84975694:/# touch /mydata/myfile
```

在主机的对应目录中，可以找到新建文件 myfile。

```
# tree
/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9
e1dcc095f35/
/var/lib/docker/volumes/1e59a53493c04b09a4caf95eecf839143d9a83db68d4216e2f6a9
e1dcc095f35/
└── _data
    └── myfile
```

删除容器时，不会删除挂载卷。在/var/lib/docker/volumes 目录下，挂载卷会一直存在。

启动容器，在/mydata 目录中新建 myfile。

```
# docker run -it test bash
root@12fbb637fa7f:/# touch /mydata/myfile
```

在主机对应的目录下，发现在容器中新建的文件 myfile。

```
# tree /var/lib/docker/volumes/
fefdf4457cc67270d85ecd0a0a18fa485a9061cfb1993a688fe396944472874f/
/var/lib/docker/volumes/
fefdf4457cc67270d85ecd0a0a18fa485a9061cfb1993a688fe396944472874f
/
└── _data
    └── myfile
```

退出容器，并通过 docker rm 删除容器。

```
root@12fbb637fa7f:/# exit
# docker rm 12fbb637fa7f
```

容器不存在，但是挂载卷依然存在。

```
# docker ps -a
CONTAINER ID        IMAGE                                                COMMAND
CREATED             STATUS                      PORTS                    NAMES
# tree /var/lib/docker/volumes/
fefdf4457cc67270d85ecd0a0a18fa485a9061cfb1993a688fe396944472874f/
/var/lib/docker/volumes/
fefdf4457cc67270d85ecd0a0a18fa485a9061cfb1993a688fe396944472874f
/
└── _data
    └── myfile
```

如果要同时删除容器和挂载卷，则必须在 docker rm 中使用-v 参数。

启动容器，在/mydata 目录中新建 myfile。

```
# docker run -it test bash
root@06828fa1c51d:/# touch /mydata/myfile
```

在主机对应的目录下，发现在容器中新建的文件 myfile。

```
# tree /var/lib/docker/volumes/
f75b896e993bf44c69de73aed4a3816da3101ff047eef943bbb487a454655c79/
```

```
/var/lib/docker/volumes/
f75b896e993bf44c69de73aed4a3816da3101ff047eef943bbb487a454655c79/
└── _data
    └── myfile
```

退出容器，并通过 docker rm 删除容器。

```
root@06828fa1c51d:/# exit
# docker rm -v 06828fa1c51d
```

容器不存在，挂载卷也被删除。

```
# docker ps -a
CONTAINER ID        IMAGE                                          COMMAND
CREATED             STATUS                    PORTS                NAMES
tree /var/lib/docker/volumes/
/var/lib/docker/volumes/

0 directories, 0 files
```

11.2.9　UESR 指令最佳实践

使用 USER 指令设置应用程序的所有者。如果容器中的应用程序运行时不需要特殊权限，则可以通过 USER 指令把应用程序的所有者设置为非 root 用户。首先在 Dockerfile 中，使用 RUN 指令创建用户和用户组。然后指定应用程序的所有者。

```
RUN groupadd -r postgres \
  && useradd -r -g postgres postgres
USER postgres
```

新建用户和用户组时，如果不指定 UID/GID，则每次编译镜像时，系统会分配不同的 UID/GID。为保持一致，应在新建用户和用户组时，指定 UID/GID。

制作镜像时，应避免安装和使用 sudo 命令。因为该命令使用的 TTY 不确定，对接收信号量也会造成影响。如果确实需要使用 sudo 功能，则可以使用 gosu 命令。例如，使用 root 用户初始化一个 Daemon，然后以非 root 用户启动这个 Daemon。

为了减少镜像体积，应避免在不同用户之间切换。

11.2.10　使用 gosu 工具

gosu 是一个使用 Golang 语言开发的工具，用于取代 shell 中的 sudo 命令。su 和 sudo 命令有一些缺陷，主要是会引起不确定的 TTY，对信号量的转发也存在问题。su 和 sudo 命令提供了丰富的用户切换功能，但是安装和使用比较些复杂。如果仅为了使用特定用户运行应用程序，使用 su 或者 sudo 显得太重了。为此，开发者开发了 gosu 这个工具。

gosu 直接借用了 libcontainer 在容器中启动应用程序的原理，使用/etc/passwd 处理应用程序。gosu 首先找出指定的用户或用户组，然后切换到该用户或用户组。接下来，

使用 exec 启动应用程序。到此为止，gosu 完成了它的工作，不会参与到应用程序后面生命周期中。使用这种方式，避免了 gosu 处理 TTY 和转发信号量，把这两个工作直接交给应用程序完成。

```
$ gosu
Usage: ./gosu user-spec command [args]
   ie: ./gosu tianon bash
       ./gosu nobody:root bash -c 'whoami && id'
       ./gosu 1000:1 id

./gosu version: 1.1 (go1.3.1 on linux/amd64; gc)
```

在 Debian 中，使用下面的指令安装 gosu。

```
FROM debian

# 通过环境变量，设置 gosu 的版本信息。
ENV GOSU_VERSION 1.7

# 安装 guso 和依赖的软件包。
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recommends ca-certificates
wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu
"https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg
--print-architecture)" \
    && wget -O /usr/local/bin/gosu.asc
"https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg
--print-architecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys
B42F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget
```

在 Alpine Linux 中，使用一下指令安装 gosu。

```
FROM alpine (3.3+)

# 设置 gosu 版本信息，在镜像中安装 gosu。
ENV GOSU_VERSION 1.7
RUN set -x \
    && apk add --no-cache --virtual .gosu-deps \
       dpkg \
       gnupg \
       openssl \
    && wget -O /usr/local/bin/gosu
"https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg
--print-architecture)" \
```

```
    && wget -O /usr/local/bin/gosu.asc
"https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$(dpkg
--print-architecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys
B42F6819007F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apk del .gosu-deps
```

11.2.11　WORKDIR 指令最佳实践

WORKDIR 指令设置 Dockerfile 中其他指令的工作目录。为了增加指令的可读性，避免出错，应在 WORKDIR 中使用绝对路径。切换工作目录时，要避免使用 RUN cd .. && do something，这种方式会造成 Dockerfile 缺乏可读性，容易引起错误，不易维护。

下例是 Docker Hub 上一个 zookeeper 镜像中使用的 WORKDIR 指令。

```
FROM java:openjdk-8-jre-alpine
MAINTAINER Justin Plock <justin@plock.net>

# 设置镜像的编译参数。
ARG MIRROR=http://apache.mirrors.pair.com
ARG VERSION=3.4.8

# 设置镜像的标签信息。
LABEL name="zookeeper" version=$VERSION

# 安装 zookeeper。
RUN apk add --no-cache wget bash \
    && mkdir /opt \
    && wget -q -O - $MIRROR/zookeeper/zookeeper-$VERSION/zookeeper-$VERSION
.tar.gz | tar -xzf - -C /opt \
    && mv /opt/zookeeper-$VERSION /opt/zookeeper \
    && cp /opt/zookeeper/conf/zoo_sample.cfg /opt/zookeeper/conf/zoo.cfg \
    && mkdir -p /tmp/zookeeper

# 导出 2181、2888 和 3888 端口。
EXPOSE 2181 2888 3888

# 切换工作目录到/opt/zookeeper。
WORKDIR /opt/zookeeper

# 导出 zookeeper 的相关目录。
VOLUME ["/opt/zookeeper/conf", "/tmp/zookeeper"]

# 设置镜像的入口程序，并通过 CMD 设置入口程序的参数。
ENTRYPOINT ["/opt/zookeeper/bin/zkServer.sh"]
```

```
CMD ["start-foreground"]
```

11.2.12　ONBUILD 指令最佳实践

ONBUILD 指令在基础镜像中设置钩子指令。在子镜像中，首先执行基础镜像中的 ONBUILD 设置的指令。在一些开发语言的镜像中有一些 ONBUILD 指令，通过这些指令，在子镜像中可以开发定制程序。

一些基础镜像会有专门的 ONBUILD 版本，如 ruby:1.9-onbuild。以下是 Ruby 官方镜像 ONBUILD 版本中的 ONBUILD 指令。

```
FROM ruby:2.1

# 通过 Gemfile.lock 文件判断 Gemfile 是否被修改。
# 一旦 Gemfile 文件被修改，就报错。
RUN bundle config --global frozen 1

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# 设置 ONBUILD 指令。
ONBUILD COPY Gemfile /usr/src/app/
ONBUILD COPY Gemfile.lock /usr/src/app/
ONBUILD RUN bundle install

ONBUILD COPY . /usr/src/app
```

在 ONBUILD 指令中使用 ADD 和 COPY 时，需要特别注意，如果子镜像的编译目录中没有源文件或文件夹，编译子镜像会失败。

如果自己开发带有 ONBUILD 指令的基础镜像，最好给这些镜像打上特殊的标签，如 onbuild。

11.3　如何减小镜像体积

制作镜像时应尽量减小镜像体积，这样可以加快镜像的上传、下载。在制作镜像时，应尽量避免安装不需要的包。在 Dockerfile 中，应避免使用 RUN apt update 或 RUN yum update 命令，这些命令会在镜像中自动下载很多不需要的文件或者软件包。

读者可能会问：“为什么一条简单的 apt/yum update 会编译出一个超大体积的镜像？”因为读者不了解 Docker 镜像制作的原理。为了帮助读者理解这个问题，在本节中，带领读者深入探讨 Docker 镜像的编译机制。同时介绍一些技巧，帮助读者在更新软件包的同时，减小镜像体积。

首先，下载 Fedora 23 的镜像。下载完毕以后，镜像的大小是 204.7 MB。

```
# docker pull fedora:23
```

```
Trying to pull repository docker.io/library/fedora ... 23: Pulling from library/fedora
a3ed95caeb02: Pull complete
236608c7b546: Pull complete
Digest: sha256:008c29c39619425de93eee20100661bef85a3f4fe0eaaf5b33532f615ccc2cd7
Status: Downloaded newer image for fedora:23
# docker images
REPOSITORY    TAG       IMAGE ID        CREATED         SIZE
fedora        23        ddd5c9c1d0f2    3 months ago    204.7 MB
```

使用 Fedora 23 作为基础镜像，制作一个包含 JDK 1.8 和 WildFly 9.0.2 Final 的自定义镜像。以下是制作镜像使用的 Dockerfile。

```
# This file is for demonstration purposes only. Please, don't use this image for
production

# 使用 fedora:23 作为基础镜像。
FROM fedora:23

# 设置作者信息。
MAINTAINER Rafael Benevides <benevides@redhat.com>

# 设置 WILDFLY 版本号。
ENV WILDFLY_VERSION 9.0.2.Final

# 安装需要的软件包。
RUN dnf -y install java-1.8.0-openjdk-devel tar && dnf clean all

# 新建 wildfly 的主目录。
RUN mkdir -p /opt/jboss/wildfly

# 下载 Wildfly 9.0.2 压缩文件。
# 解压后，移动到/opt/jboss/wildfly 目录中。
RUN cd /tmp \
  && curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz \
     && tar xf wildfly-$WILDFLY_VERSION.tar.gz \
     && mv /tmp/wildfly-$WILDFLY_VERSION /opt/jboss/wildfly \
     && rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz

# 导出 8080 端口。
EXPOSE 8080

# 设置镜像的启动程序。
# 启动容器时，将以 standalone 模式运行 WildFly，并绑定所有端口。
CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0"]
```

使用 docker build 命令编译镜像，镜像名为 wildfly:9.0.2。

```
# docker build -t wildfly:9.0.2 .
Sending build context to Docker daemon 58.88 kB
Step 1 : FROM fedora:23
```

```
 ---> ddd5c9c1d0f2
Step 2 : MAINTAINER Rafael Benevides <benevides@redhat.com>
 ---> Running in 1c056feaadda
 ---> f3f8d3423bf4
Removing intermediate container 1c056feaadda
Step 3 : ENV WILDFLY_VERSION 9.0.2.Final
 ---> Running in d8e89b0668d8
 ---> 83aafed3e4be
Removing intermediate container d8e89b0668d8
Step 4 : RUN dnf -y install java-1.8.0-openjdk-devel tar && dnf clean all
 ---> Running in ed247a3311c4
Last metadata expiration check performed 0:00:20 ago on Wed Jun 22 09:07:06 2016.
Dependencies resolved.
......
Complete!
Cleaning repos: fedora updates
Cleaning up Everything
 ---> a52f7477a49f
Removing intermediate container ed247a3311c4
Step 5 : RUN mkdir -p /opt/jboss/wildfly
 ---> Running in 85052a788da6
 ---> 35dfbcc49699
Removing intermediate container 85052a788da6
Step 6 : RUN cd /tmp  && curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz
&& tar xf wildfly-$WILDFLY_VERSION.tar.gz     && mv /tmp/wildfly-$WILDFLY_VERSION
/opt/jboss/wildfly      && rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz
 ---> Running in 1e12f894cdcb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  129M  100  129M       0     0   284k      0  0:07:45  0:07:45 --:--:--  332k
 ---> c20b9bb1fc30
Removing intermediate container 1e12f894cdcb
Step 7 : EXPOSE 8080
 ---> Running in 31ad6d7905b2
 ---> 5f6c7d1c28fb
Removing intermediate container 31ad6d7905b2
Step 8 : CMD /opt/jboss/wildfly/bin/standalone.sh -b 0.0.0.0
 ---> Running in 761cc7199b3a
 ---> b27e27e9afc5
Removing intermediate container 761cc7199b3a
Successfully built b27e27e9afc5
```

镜像创建完成后体积已经达到了 576.2MB。

```
# docker images
REPOSITORY   TAG    IMAGE ID       CREATED       SIZE
wildfly          9.0.2   b27e27e9afc5    3 minutes ago    576.2 MB
```

查看每层的大小。在本例的镜像中 JDK 1.8 有 212.5MB，WildFly 有 159.1 MB，体

积相当大。

```
# docker history wildfly:9.0.2
IMAGE            CREATED           CREATED BY
SIZE             COMMENT
b27e27e9afc5     5 minutes ago     /bin/sh -c #(nop) CMD ["/opt/jboss/wildfly/bi  0 B
5f6c7d1c28fb     5 minutes ago     /bin/sh -c #(nop) EXPOSE 8080/tcp              0 B
c20b9bb1fc30     5 minutes ago     /bin/sh -c cd /tmp  && curl -O https://downl
 159.1 MB
35dfbcc49699     12 minutes ago    /bin/sh -c mkdir -p /opt/jboss/wildfly    0 B
a52f7477a49f     13 minutes ago    /bin/sh -c dnf -y install java-1.8.0-openjdk-
212.5 MB
83aafed3e4be     17 minutes ago    /bin/sh -c #(nop) ENV WILDFLY_VERSION=
9.0.2.F   0 B
f3f8d3423bf4     17 minutes ago    /bin/sh -c #(nop) MAINTAINER Rafael
Benevides   0 B
ddd5c9c1d0f2     3 months ago      /bin/sh -c #(nop) ADD file:bcb5e5cddd4c4d1cac
204.7 MB
<missing>        3 months ago      /bin/sh -c #(nop) MAINTAINER Patrick
Uiterwij   0 B
```

当 wildFly 有更新时，需要升级镜像，如升级到 wildfly:10.0.0.final。大部分开发者为了节约时间，不想重新安装 JDK 1.8。他们会选择 wildfly:9.0.2 作为基础镜像，用 wildfly:10.0.0.final 替换 wildfly:9.0.2.final。

```
#This file is for demonstration purposes only. Please, don't use this image for
production

# 使用 wildfly 9.0.2 作为基础镜像。
FROM wildfly:9.0.2

# 设置作者信息。
MAINTAINER Rafael Benevides <benevides@redhat.com>

# 设置 WildFly 的版本号。
ENV WILDFLY_VERSION 10.0.0.Final

# 删除镜像中原来的/opt/jboss/wildfly 目录。
RUN rm -rf /opt/jboss/wildfly/*

# 下载 Wildfly 10.0.0 压缩包。
# 解压后，移动到/opt/jboss/wildfly 目录中。
RUN cd /tmp \
  && curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz \
     && tar xf wildfly-$WILDFLY_VERSION.tar.gz \
     && mv /tmp/wildfly-$WILDFLY_VERSION /opt/jboss/wildfly \
     && rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz

# 导出 8080 端口。
```

```
EXPOSE 8080

# 设置镜像的启动程序。
# 启动容器时，将以 standalone 模式运行 WildFly，并绑定所有端口。
CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0"]
```

重新编译镜像，得到 wildfly:10.0.0。

```
# docker build -t wildfly:10.0.0 .
Sending build context to Docker daemon 60.42 kB
Step 1 : FROM wildfly:9.0.2
 ---> b27e27e9afc5
Step 2 : MAINTAINER Rafael Benevides <benevides@redhat.com>
 ---> Running in a70bd1b74a46
 ---> 78171d47aa4a
Removing intermediate container a70bd1b74a46
Step 3 : ENV WILDFLY_VERSION 10.0.0.Final
 ---> Running in 134283e9329f
 ---> d06aa3958ec4
Removing intermediate container 134283e9329f
Step 4 : RUN rm -rf /opt/jboss/wildfly/*
 ---> Running in 96d92d55c58c
 ---> 4d982f4d8f99
Removing intermediate container 96d92d55c58c
Step 5 : RUN cd /tmp  && curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz      && tar xf wildfly-$WILDFLY_VERSION.tar.gz      && mv /tmp/wildfly-
$WILDFLY_VERSION /opt/jboss/wildfly     && rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz
 ---> Running in b613fcc613ff
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  130M  100  130M    0     0   275k      0  0:08:06  0:08:06 --:--:--  258k
 ---> 5a439dfe2ade
Removing intermediate container b613fcc613ff
Step 6 : EXPOSE 8080
 ---> Running in 7f92f7b58c47
 ---> c38ba363efbd
Removing intermediate container 7f92f7b58c47
Step 7 : CMD /opt/jboss/wildfly/bin/standalone.sh -b 0.0.0.0
 ---> Running in bd86e10a6fe6
 ---> d15e50a74c8f
Removing intermediate container bd86e10a6fe6
Successfully built d15e50a74c8f
```

使用这种方式升级 wildfly 后，镜像的体积变为 737.1MB，而不是 567MB。

```
# docker images
REPOSITORY   TAG      IMAGE ID       CREATED          SIZE
wildfly      10.0.0   d15e50a74c8f   2 minutes ago    737.1 MB
```

即使在镜像中删除 wildfly 9.0.2.final，新镜像的体积也会比基础镜像大 160MB。增加的空间并不是两个版本之间的差异造成的。

为什么镜像体积会增加这么多呢？原因是 Docker 在编译镜像时，使用了写时复制（Copy On Wirte）技术。这项技术可以加快容器的启动时间，与使用 hypervisor 技术的虚拟机相比，启动一个容器只需要 10 几秒。虽然写时复制技术提高了容器的运行效率，但会占用额外的磁盘空间。制作镜像时，开发者需要考虑多方面的因素，减少镜像体积。

Dockerfile 中的每条指令都会生成一个镜像层，每个镜像层都会占用一些磁盘空间。在镜像中升级 wildfly，实际上创建了一个新的镜像层，并不是简单地替换旧版本的 wildfly。为了减少镜像层的数量，开发者应在同一条 RUN 指令中执行文件操作，如移动文件、释放文件、删除文件。

下例展示了如何在一条 RUN 指令中，下载 wildfly 源码、解压文件、移动文件、删除压缩包。

```
RUN cd /tmp \
  && curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz \
  && tar xf wildfly-$WILDFLY_VERSION.tar.gz \
  && mv /tmp/wildfly-$WILDFLY_VERSION /opt/jboss/wildfly \
  && rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz
```

如果把每条文件操作命令放在单独的 RUN 指令中，就会增加额外的 4 个镜像层，容器体积也会多增加 300MB 左右。

```
#Add Wildfly 10.0.0 to this image
WORKDIR /tmp
RUN curl -O
https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-$WILDFLY_VERSION.
tar.gz
RUN tar xf wildfly-$WILDFLY_VERSION.tar.gz
RUN mv /tmp/wildfly-$WILDFLY_VERSION /opt/jboss/wildfly
RUN rm /tmp/wildfly-$WILDFLY_VERSION.tar.gz
```

在 Dockerfile 中，一般使用 apt-get update 或 yum update 更新软件列表，然后安装需要的程序和软件包。应该在一条 RUN 指令中，更新安装源、安装程序、清理缓存，这样可以减少镜像体积。

下面是使用 apt-get 安装软件的例子。

```
RUN apt-get update && apt-get install -y \
    vim \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
```

下面是使用 yum 安装软件的例子。

```
RUN yum update && yum -y install \
    vim \
  && yum clean all
```

11.4　一些官方镜像的 Dockerfile

前面介绍的都是开发 Dockerfile 的一些技巧，很多官方镜像的制作都使用了这些技巧。下面从 Docker Hub 上选取一些官方镜像的 Dockerfile，学习如何在实践中应用这些技巧。

11.4.1　Golang 镜像

Golang 是 google 开发的一门静态类型编程语言。Golang 语言从 C 语言演化而来，在 C 语言的基础上，添加了一些高级功能，如垃圾回收，更丰富的内置类型，函数多返回值，匿名函数和闭包，反射等。

Golang 镜像有不同的标签，应用在不同场景下。

1. golang:1.6.2

这类镜像是默认的镜像。当不确定需求时，可以使用这个镜像。这个镜像在容器中挂载用户源码，编译程序，并运行程序。这类镜像是基于 buildpack-deps 基础镜像，buildpack-deps 包含了通常开发所必须的头文件和工具。

```
FROM buildpack-deps:jessie-scm

# 下载 gcc 及相关包。
RUN apt-get update && apt-get install -y --no-install-recommends \
    g++ \
    gcc \
    libc6-dev \
    make \
  && rm -rf /var/lib/apt/lists/*

# 设置 GoLang 版本和相关信息。
ENV GOLANG_VERSION 1.6.2
ENV GOLANG_DOWNLOAD_URL
https://golang.org/dl/go$GOLANG_VERSION.linux-amd64.tar.gz
ENV GOLANG_DOWNLOAD_SHA256
e40c36ae71756198478624ed1bb4ce17597b3c19d243f3f0899bb5740d56212a

#  下载 golang 压缩包，并解压。
RUN curl -fsSL "$GOLANG_DOWNLOAD_URL" -o golang.tar.gz \
  && echo "$GOLANG_DOWNLOAD_SHA256  golang.tar.gz" | sha256sum -c - \
  && tar -C /usr/local -xzf golang.tar.gz \
  && rm golang.tar.gz

# 设置 GOPATH，并把 GOPATH 添加到系统 PATH 环境变量中。
ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

# 修改 GoLang 目录的属性。
```

```
RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH"
WORKDIR $GOPATH

# 复制 go-wrapper 到镜像中。
COPY go-wrapper /usr/local/bin/
```

2. golang:1.6.2-wheezy

这类镜像与默认镜像相似，区别在于使用 Debian 的 Wheezy 版本。如果需要使用 Wheezy 版本的 Debian 作为 OS，可以使用这个版本。

```
FROM buildpack-deps:wheezy-scm

# 下载 gcc 及相关包。
RUN apt-get update && apt-get install -y --no-install-recommends \
    g++ \
    gcc \
    libc6-dev \
    make \
  && rm -rf /var/lib/apt/lists/*

# 设置 GoLang 版本和相关信息。
ENV GOLANG_VERSION 1.6.2
ENV GOLANG_DOWNLOAD_URL
https://golang.org/dl/go$GOLANG_VERSION.linux-amd64.tar.gz
ENV GOLANG_DOWNLOAD_SHA256
e40c36ae71756198478624ed1bb4ce17597b3c19d243f3f0899bb5740d56212a

#  下载 Golang 压缩包，并解压。
RUN curl -fsSL "$GOLANG_DOWNLOAD_URL" -o golang.tar.gz \
  && echo "$GOLANG_DOWNLOAD_SHA256  golang.tar.gz" | sha256sum -c - \
  && tar -C /usr/local -xzf golang.tar.gz \
  && rm golang.tar.gz

# 设置 GOPATH，并把 GOPATH 添加到系统 PATH 环境变量中。
ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

# 修改 GoLang 目录的属性。
RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH"
WORKDIR $GOPATH

# 复制 go-wrapper 到镜像中。
COPY go-wrapper /usr/local/bin/
```

3. golang:1.6.2-alpine

这类镜像使用 Apline Linux 作为基础镜像。Apline Linux 是目前最小的基础镜像，存储空间小于 5MB。

推荐使用该镜像，因为它占用的空间最小。需要注意的是，Apline Linux 使用 musl libc 替代 glibc。一些依赖 libc 的程序在使用该镜像的容器中运行可能会失败。幸运的是，大部分程序都能够在使用该镜像的容器中运行。

为了节约空间，Alpine Linux 镜像不会包含 git、bash 等工具。制作镜像时，需要添加对应的工具。

```
FROM alpine:3.4

# 设置 GoLang 版本和相关信息。
ENV GOLANG_VERSION 1.6.2
ENV GOLANG_SRC_URL https://golang.org/dl/go$GOLANG_VERSION.src.tar.gz
ENV GOLANG_SRC_SHA256
787b0b750d037016a30c6ed05a8a70a91b2e9db4bd9b1a2453aa502a63f1bccc

# 针对 https://golang.org/issue/14851 问题，复制补丁文件到镜像中。
COPY no-pic.patch /

# 安装 Golang 依赖的软件包。
# 下载 Golang 源码，并编译。
RUN set -ex \
  && apk add --no-cache --virtual .build-deps \
    bash \
    ca-certificates \
    gcc \
    musl-dev \
    openssl \
    go \
  \
  && export GOROOT_BOOTSTRAP="$(go env GOROOT)" \
  \
  && wget -q "$GOLANG_SRC_URL" -O golang.tar.gz \
  && echo "$GOLANG_SRC_SHA256  golang.tar.gz" | sha256sum -c - \
  && tar -C /usr/local -xzf golang.tar.gz \
  && rm golang.tar.gz \
  && cd /usr/local/go/src \
  && patch -p2 -i /no-pic.patch \
  && ./make.bash \
  \
  && rm -rf /*.patch \
  && apk del .build-deps

# 设置 GOPATH，并把 GOPATH 添加到系统 PATH 环境变量中。
ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

# 修改 GoLang 目录的属性。
RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH"
WORKDIR $GOPATH
```

4. golang:1.6.2-onbuild

开发者使用这类镜像作为基础镜像生成子镜像。

在实践中，仅在开发阶段使用这类镜像。因为它可以快速生成二进制文件。不推荐在长期运行的环境中使用该类镜像。因为开启 ONBUILD 指令后，对镜像会失去完全控制。

一旦掌握了如何使用 Docker 运行 Golang 程序，推荐使用非 onbuild 标签镜像作为基础镜像，这样开发者就可以完全控制自己的镜像，其他人也可以通过 Dockerfile 明白该镜像在做什么。

```
FROM golang:1.6

# 建立 Golang 的工作目录，并进入工作目录。
RUN mkdir -p /go/src/app
WORKDIR /go/src/app

# 设置镜像的启动程序。
CMD ["go-wrapper", "run"]

# 添加 ONBUILD 指令。
ONBUILD COPY . /go/src/app
ONBUILD RUN go-wrapper download
ONBUILD RUN go-wrapper install
```

11.4.2　Perl 镜像

Perl 是一种常用的脚本语言，在超过 100 种计算机平台上运行，适用广泛。它借鉴了很多其他语言的优点，包括 C、Shell、awk、sed 等。

perl:5.24 镜像提供了 perl:5.24 的运行环境。在 Dockerfile 中，下载 Perl 源文件、编译工具，通过 build 完成编译。开发者可以使用该镜像运行 perl 脚本，也可以从该镜像生成定制化的子镜像。

```
FROM buildpack-deps

# 设置作者信息。
MAINTAINER Peter Martini <PeterCMartini@GMail.com>

# 安装需要的软件包。
RUN apt-get update \
    && apt-get install -y curl procps \
    && rm -fr /var/lib/apt/lists/*

# 新建/usr/src/perl 目录。
RUN mkdir /usr/src/perl

# 复制补丁文件到/usr/src/perl 目录。
```

```
COPY *.patch /usr/src/perl/

# 切换工作目录到/usr/src/perl 目录。
WORKDIR /usr/src/perl

# 下载 perl 源码，并编译。
RUN curl -SL https://cpan.metacpan.org/authors/id/R/RJ/RJBS/perl-5.24.0.tar.bz2 -o
perl-5.24.0.tar.bz2 \
    && echo '298fa605138c1a00dab95643130ae0edab369b4d *perl-5.24.0.tar.bz2' |
    sha1sum -c - \
    && tar --strip-components=1 -xjf perl-5.24.0.tar.bz2 -C /usr/src/perl \
    && rm perl-5.24.0.tar.bz2 \
    && cat *.patch | patch -p1 \
    && ./Configure -Duse64bitall -Duseshrplib  -des \
    && make -j$(nproc) \
    && TEST_JOBS=$(nproc) make test_harness \
    && make install \
    && cd /usr/src \
    && curl -LO https://raw.githubusercontent.com/miyagawa/cpanminus/master/cpanm \
    && chmod +x cpanm \
    && ./cpanm App::cpanminus \
    && rm -fr ./cpanm /root/.cpanm /usr/src/perl /tmp/*

# 切换工作目录到/root 目录。
WORKDIR /root

# 设置镜像的启动程序。
CMD ["perl5.24.0","-de0"]
```

11.4.3　Hy 镜像

Hy 是一种 LISP 语言，用于把表达式翻译成 Python 的抽象语法树。开发者既可以使用该镜像运行 Hy 程序，也可以使用该镜像作为 Hy 工程的基础镜像。

```
# Base image
#
# VERSION   0.2
FROM       python:3
MAINTAINER  Paul R. Tagliamonte <paultag@hylang.org>

# 复制文件到/opt/hylang/hy 目录。
ADD . /opt/hylang/hy

# 安装 hy 到/opt/hylang/hy 目录。
RUN pip3 install -e /opt/hylang/hy

# 设置镜像的启动程序。
CMD ["hy"]
```

11.4.4　Rails 镜像

Rails 是一个用 Ruby 开发的 Web 框架。开发者可以使用该镜像生成一个 Web 服务器。

1．rails:4.2.6

该镜像提供一个 Rails 的基本框架，包括 Ruby、Rails、NodeJS、MySQL Client、Sqlit3 和 Postgresql Client。

```
FROM ruby:2.3

# 安装 nodejs。
RUN apt-get update && apt-get install -y nodejs --no-install-recommends && rm -rf
/var/lib/apt/lists/*

# 安装 mysql-client，postgresql-client 和 sqlite3。
RUN apt-get update && apt-get install -y mysql-client postgresql-client sqlite3
--no-install-recommends && rm -rf /var/lib/apt/lists/*

# 设置 RAILS 版本。
ENV RAILS_VERSION 4.2.6

# 安装 rails。
RUN gem install rails --version "$RAILS_VERSION"
```

2．rails:onbuild

使用 onbuild 标签的镜像生成子镜像时，需要保证在编译目录中有 gemfile 和 gemfile.lock。编译自镜像成功后，将导出 3000 端口，提供 Web 服务。

```
FROM ruby:2.3

# 通过 Gemfile.lock 文件判断 Gemfile 是否被修改。
# 一旦 Gemfile 文件被修改，就报错。
RUN bundle config --global frozen 1

# 创建工作目录。
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# 设置 ONBUILD 指令。
ONBUILD COPY Gemfile /usr/src/app/
ONBUILD COPY Gemfile.lock /usr/src/app/
ONBUILD RUN bundle install

ONBUILD COPY . /usr/src/app

# 安装 nodejs，mysql-client，postgresql-client 和 sqlite3。
RUN apt-get update && apt-get install -y nodejs --no-install-recommends && rm -rf
/var/lib/apt/lists/*
```

```
RUN apt-get update && apt-get install -y mysql-client postgresql-client sqlite3
--no-install-recommends && rm -rf /var/lib/apt/lists/*

# 导出 3000 端口。
EXPOSE 3000

# 设置镜像的启动命令。
CMD ["rails", "server", "-b", "0.0.0.0"]
```

11.5　习题

本章详细介绍了使用 Dockerfile 制作镜像的最佳实践。接下来，通过习题和实验检验本章的学习成果。

（1）使用 Ubuntu 14.04 作为基础镜像，制作自己的镜像。使用 RUN 指令，在镜像中安装 curl、vim、sshd 软件包，并清理临时文件。

使用 docker build 命令编译镜像，启动容器，进入容器检查这三个软件包是否安装成功。

（2）在习题 1 的基础上，添加入口程序 start.sh。start.sh 为脚本，在脚本中设置 root 账户密码为 ghostcloud，并打印“The root password is ghostcloud”。使用 COPY 指令，把 start.sh 复制到/目录中。使用 RUN 指令，为 start.sh 脚本添加执行权限 x。使用 ENTRYPOINT 设置入口程序为 start.sh。

使用 docker build 命令编译镜像，启动容器。检查容器启动后，是否在终端中打印“The root password is ghostcloud”。进入容器检查该脚本是否存在。

（3）在习题 2 的基础上，添加导出端口，并在入口程序中启动 sshd。使用 EXPOSE 指令，导出容器 22 端口。修改 start.sh 脚本，在最后添加“service sshd start”。

使用 docker build 命令编译镜像。启动容器时，把容器 22 端口导出到主机 2222 上。在另外一台主机上，通过 SSH 登录到容器。SSH 使用 2222 作为端口，登录用户为 root，密码为 ghostcloud。检查是否能够成功登录容器。

第 12 章　使用容器提供服务

Docker Hub 上有成千上万的镜像，为开发者提供了各种服务。通过容器的方式，开发者可以快速地构建出开发、测试、生产环境，免去了搭建环境的烦琐工作。本书选取了一些 Docker Hub 上流行的官方镜像。为大家详细介绍如何快速使用这些镜像提供服务。

12.1　使用容器提供数据库服务

在本节中将介绍如何使用容器提供各种数据库服务。

12.1.1　使用容器提供 MySQL

MySQL 是目前最流行的开源关系型数据库，由瑞典 MySQL AB 公司开发，现在属于 Oracle 旗下公司。在 Web 应用中，MySQL 是最好的关系数据库之一，中小型网站的开发一般都选择 MySQL 作为网站数据库。在关系数据库中，数据保存在不同的表中，而不是将所有数据放在一个大表中，这样就提高了速度和灵活度。MySQL 所使用的 SQL 语言是用于访问数据库的最常用标准化语言。

1．镜像名

在 Docker Hub 中，MySQL 的官方镜像是 mysql，可以通过下面的命令下载镜像。

```
docker pull mysql
```

2．支持的版本

在镜像中，不同版本使用不同 tag 区分，同一版本可能有不同 tag，支持的版本如下。

- 5.7.13、5.7、5、latest；
- 5.6.31、5.6；
- 5.5.50、5.5。

3. 使用镜像

使用下面的命令启动 MySQL 容器，提供数据库服务。

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

使用--name some-mysql 设置容器名。通过-e MYSQL_ROOT_PASSWORD=my-secret-pw。设置数据库密码。tag 对应了使用的 MySQL 版本。

使用下面的命令启动 MySQL 容器，在容器中运行 MySQL 客户端命令。

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec mysql
-h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" -uroot
-p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
```

使用下面的命令进入容器，在容器中查看 MySQL 容器的运行状态。some-mysql 是运行状态容器的名字。

```
$ docker exec -it some-mysql bash
```

启动 MySQL 容器后，MySQL 的日志会在终端打印，可以通过下面命令查看 MySQL 的日志。

```
$ docker logs some-mysql
```

4. 配置 MySQL 容器

在容器中使用 MySQL 配置文件，可以通过挂载卷的方式实现。在容器中，MySQL 的配置主文件是/etc/mysql/my.cnf，其他配置文件放在/etc/mysql/conf.d 目录下。如果需要修改 MySQL 配置文件，可以使用宿主机上的配置文件覆盖容器中的文件。

在宿主机上建立/my/custom 目录，其中保存定制化的 MySQL 配置文件。把该目录挂载到容器上，覆盖/etc/mysql/conf.d 目录。启动容器后，将使用宿主机上的 MySQL 配置文件。

```
$ docker run --name some-mysql -v /my/custom:/etc/mysql/conf.d -e
MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

如果在宿主机上，开启了 SELinux 功能，挂载宿主机的配置文件可能会有问题。此时，需要为宿主机的/my/custom 目录设置 SELinux 规则。

```
$ chcon -Rt svirt_sandbox_file_t /my/custom
```

MySQL 容器的入口程序是 mysqld。可以在启动容器时追加参数，配置容器中的 MySQL 数据库。这种方式很灵活，不需要配置文件。如果想把容器中 MySQL 的编码方式修改为 UTF-8，可以使用下面的命令。

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
--character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
```

可以使用下面的命令，查看启动 mysql 容器时可以添加的选项。

```
$ docker run -it --rm mysql:tag --verbose --help
```

5. 环境变量

在 Docker 中，一般可以使用环境变量配置应用程序。一般来说，环境变量比配置文件的优先级高。环境变量可以覆盖配置文件中的同名参数，这种方式比配置文件更灵活。启动容器时，使用 docker run -e ENVIRONMENT 设置容器中的环境变量。

在 MySQL 镜像中，有一些预设的环境变量，可以通过这些环境变量，配置 MySQL。

6. MYSQL_ROOT_PASSWORD

必选参数。设置数据库 root 账户的密码。启动容器时必须设置该变量，否则会报错。

7. MYSQL_DATABASE

可选参数。设置容器启动时，新建数据库的名字。如果同时设置了 MYSQL_USER 和 MYSQL_PASSWORD 变量，则 MYSQL_USER 可以以超级用户的身份访问 MYSQL_DATABASE。

8. MYSQL_USER 和 MYSQL_PASSWORD

可选参数，必须同时设置这两个参数。设置容器启动时，新建用户的用户名和密码。该用户可以以超级用户的身份访问 MYSQL_DATABASE。

不需要使用这两个变量设置 root 的密码，root 用户默认使用 MYSQL_ROOT_PASSWORD 中设置的密码。

9. MYSQL_ALLOW_EMPTY_PASSWORD

可选参数。设置为 YES 时，允许数据库不使用密码。一般情况下，不要设置 MYSQL_ALLOW_EMPTY_PASSWORD 为 YES。一旦开启空密码模式，任何人都可以访问该数据库。

10. 数据存储

在 Docker 中有多种方式保存数据库文件，开发者必须熟悉这些数据保存方式。

11. 数据库文件保存在容器中

此时，Docker 管理容器中的数据库文件。Docker 会使用内部机制把数据库文件保存在宿主机的文件系统中。使用不同的存储驱动，文件保存的位置不同。这种方式是默认的数据库文件存储方式，简单，并且对用户透明。缺点是很难在宿主机的文件系统中定位这些文件，而且容器删除后，这些数据库文件可能也被删除。

12. 使用挂载卷做数据持久化

在宿主机上创建一个目录，通过挂载卷的方式，在容器中使用该目录。所有数据库文件都保存在宿主机的指定目录中。使用这种方式，开发者可以方便地在宿主机上定位数据库文件，而且删除容器后，这些文件继续保存在宿主机上。

通过下面的方法，可以在容器中使用挂载卷做数据持久化。

（1）在宿主机上，创建目录。

```
$ sudo mkdir -p /my/own/datadir
```

（2）启动容器时，通过-v 参数，把该目录挂载到容器中。此时，数据库文件保存在宿主机的/my/own/datadir 中。

```
$ docker run --name some-mysql -v /my/own/datadir:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

13. 备份数据库

可以使用容器，备份 MySQL 的数据库。最简单的方法是使用 docker exec 命令。使用下面命令，可以把容器中的数据库备份到宿主机上。

```
$ docker exec some-mysql sh -c 'exec mysqldump --all-databases -uroot
-p"$MYSQL_ROOT_PASSWORD"' > /some/path/on/your/host/all-databases.sql
```

12.1.2　使用容器提供 MongoDB

Mongo 名称来自于 Humongous，是非关系型数据库。MongoDB 是一种文档导向数据库，可以跨平台运行。由 C++撰写而成，以此来解决应用程序开发社区中的大量现实问题。与传统数据库不用，MongoDB 使用表结构。MongoDB 使用 BSON 文件动态保存 Schema，BSON 是一种类似 JSON 的数据格式，这样可以方便地添加、修改数据库中的数据项。

MongoDB 诞生于 2007 年 10 月，由 10gen 团队开发。2009 年 2 月 10gen 把该项目开源。此后，许多大型网站开始使用 MongoDB 作为数据库，包括 Craigslist、eBay、Foursquare、SourceForge、Viacom 和 New York Times 等。MongoDB 是目前最流行的非关系型数据库之一。

1. 镜像名

在 Docker Hub 中，MongoDB 的官方镜像是 mongo，可以通过下面的命令下载镜像。

```
docker pull mongo
```

2. 支持的版本

在镜像中，不同版本使用不同 tag 区分，同一版本可能有不同 tag，支持的版本如下。

- 2.6.12、2.6, 2;
- 3.0.12、3.0;
- 3.2.7、3.2、3、latest;
- 3.3.9、3.3。

3. 使用镜像

启动一个 mongo 容器，使用 MongoDB 的默认端口 27017 对外提供服务。

```
$ docker run --name some-mongo -p 27017:27017 -d mongo
```

通过 link 方式，把另一个容器与 mongo 容器连接起来。some-web 容器可以使用 mongo 容器提供的数据库功能。

```
$ docker run --name some-web --link some-mongo:mongo -d web
```

启动 mongo 容器作为客户端访问 MongoDB 数据库。

```
$ docker run -it --link some-mongo:mongo --rm mongo sh -c 'exec mongo
"$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/test"'
```

4. 认证和授权

MongoDB 默认不开启认证模式。

（1）使用下面的命令在 mongo 容器中开启认证模式。

```
$ docker run --name some-mongo -d mongo -auth
```

（2）添加管理员用户。

```
$ docker exec -it some-mongo mongo admin
connecting to: admin
\> db.createUser({ user: 'jsmith', pwd: 'some-initial-password', roles: [ { role:
"userAdminAnyDatabase", db: "admin" } ] });
Successfully added user: {
    "user" : "jsmith",
    "roles" : [
        {
            "role" : "userAdminAnyDatabase",
            "db" : "admin"
        }
    ]
}
```

（3）连接数据库。

```
$ docker run -it --rm --link some-mongo:mongo mongo mongo -u jsmith -p
some-initial-password --authenticationDatabase admin some-mongo/some-db
\> db.getName();
some-db
```

5. 数据存储

在 Docker 中有多种方式保存数据库文件，开发者必须熟悉这些数据保存方式。

6. 数据库文件保存在容器中

此时，Docker 管理容器中的数据库文件。Docker 会使用内部机制把数据库文件保存在宿主机的文件系统中。使用不同的存储驱动，文件保存的位置不同。这种方式是默认的数据库文件存储方式，简单，并且对用户透明。缺点是很难在宿主机的文件系

统中定位这些文件，而且容器删除后，这些数据库文件可能也被删除。

7．使用挂载卷做数据持久化

在宿主机上创建一个目录，通过挂载卷的方式在容器中使用该目录。所有数据库文件都保存在宿主机的指定目录中。使用这种方式，开发者可以方便地在宿主机上定位数据库文件，而且删除容器后，这些文件继续保存在宿主机上。

注意　Docker 原生仅支持 Linux 平台。在 Windows 和 Mac OS 中，Docker 使用 VirtualBox 虚拟机运行 Docker Daemon。MongoDB 使用内存映射数据库文件，所以不支持 VirtualBox 的共享目录机制。在 Windows 和 Mac OS 平台上，运行 mongo 容器，不能使用挂载卷方式。

通过下面的方法，可以在容器中使用挂载卷做数据持久化。

（1）在宿主机上，创建目录。

```
$ sudo mkdir -p /my/own/datadir
```

（2）启动容器时，通过-v 参数把该目录挂载到容器中。此时，数据库文件保存在宿主机的/my/own/datadir 中。

```
$ docker run --name some-mongo -v /my/own/datadir:/data/db -d mongo:tag
```

如果宿主机开启 SELinux，启动 mongo 容器可能会有问题。目前的补救方法是为宿主机的/my/own/datadir 目录分配相应的 SELinux 策略。

```
$ chcon -Rt svirt_sandbox_file_t /my/own/datadir
```

12.2　如何使用容器提供 Web 服务

在本节中将介绍如何使用容器提供各种 Web 服务。

12.2.1　使用容器提供 Apache HTTP 服务

Apache HTTP Server，简称 Apache，是 Apache 基金会开发的一个开放源服务器，可以在大多数操作系统中运行。由于其具有的跨平台性和安全性，被广泛使用，是最流行的 Web 服务器端软件之一。

1．镜像名

在 Docker Hub 中，Apache HTTP 的官方镜像是 httpd，可以通过下面的命令下载镜像。

```
docker pull httpd
```

2．支持的版本

在镜像中，不同版本使用不同 tag 区分，同一版本可能有不同 tag，支持的版本如下。

- 2.2.31、2.2；
- 2.4.20、2.4、2、latest。

3．使用镜像

httpd 镜像中只包含了 Apache HTTP 服务器本身，没有安装 PHP 等程序。如果希望使用安装了 PHP 的 Apache HTTP 服务，可以使用 PHP:apache 镜像。

启动一个 Apache 容器，提供 Apache HTTP 服务器。需要导出容器中的 80 端口。服务器的内容需要放在容器中/usr/local/apache2/htdocs/目录下，可以使用挂载卷方式关联服务器内容到容器中。

```
$ docker run -d --name my-apache -p 80:80 -v /my/public-html/:/usr/local/apache2/
htdocs/ httpd
```

也可以使用 httpd 镜像作为基础镜像，定制自己的服务器镜像。

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
```

使用 docker build 命令编译镜像。启动自定义镜像，提供 Apache HTTP 服务。

```
$ docker build -t my-apache2 .
$ docker run -it --rm --name my-running-app my-apache2
```

4．配置 httpd 容器

在容器中，Apache HTTP 的配置文件是/usr/local/apache2/conf/httpd.conf。开发者可以直接进入容器修改该文件。

```
$ docker exec -it my-apache bash
root@12fbb637fa7f:/# vi /usr/local/apache2/conf/httpd.conf
```

也可以在宿主机上创建配置文件，然后挂载到容器中。

```
$ vi /my-apache/httpd.conf
$ docker run -d --name my-apache -p 80:80 -v /my-apache/httpd.conf:
/usr/local/apache2/conf/httpd.conf
```

12.2.2　使用容器提供 Django 服务

Django 是一个开源的 Web 应用框架，使用 Python 作为开发语言，采用了 MVC 的设计模式。Django 最初用于管理劳伦斯出版集团旗下的一些以新闻网站，于 2005 年 7 月开源，使用 BSD 许可证书发布。这套框架是以比利时的吉普赛爵士吉他手 Django Reinhardt 的名字来命名的。Django 的设计初衷是快速建立复杂的，以数据库为中心的网站。使用该框架后，开发者可以方便地加入不同组件，快速开发出复杂的网站。

1. 镜像名

在 Docker Hub 中，Django 的官方镜像是 django，可以通过下面的命令下载镜像。

```
docker pull django
```

2. 支持的版本

- 1.9.7-python3、1.9-python3、1-python3、python3、1.9.7、1.9、1、latest；
- python3-onbuild、onbuild；
- 1.9.7-python2、1.9-python2、1-python2、python2；
- python2-onbuild。

3. 使用镜像

可以直接启动 Django 容器，提供 Web 服务。

```
$ docker run --name some-django-app -v "$PWD":/usr/src/app -w /usr/src/app -p
8000:8000 -d django bash -c "pip install -r requirements.txt && python manage.py
runserver 0.0.0.0:8000"
```

也可以使用 django:onbuild 作为基础镜像，定制自己的 Django 镜像。在 Django 的 app 工程目录的根目录中新建 Dockerfile。

```
FROM django:onbuild
```

django:onbuild 有一些默认指令，在编译定制 Django 镜像时，会执行这些指令，包括 COPY . /usr/src/app、RUN pip install、EXPOSE 8000。并设置镜像中的 CMD 为 python manage.py runserver。

编译镜像，并启动容器。

```
$ docker build -t my-django-app .
$ docker run --name some-django-app -d my-django-app
```

启动容器时可以指定导出端口，如 8000。这样就可以在浏览器中，通过 http://container-ip:8000 访问 Django 网站。

```
$ docker run --name some-django-app -p 8000:8000 -d my-django-app
```

12.2.3　使用容器提供 Gitlab 服务

GitLab 是一个用于代码管理的开源项目，可以提供类似 github.com 的代码管理系统。使用 Git 作为代码管理工具，并提供权限认证、bug 追踪、wiki 等功能。

1. 镜像名

Gitlab 分为个人用户版（Community Edition）和企业版（Enterprise Edition）。在 Docker Hub 中，Gitlab 个人用户版的官方镜像是 gitlab/gitlab-ce，可以通过下面的命令下载镜像。

```
docker pull gitlab/gitlab-ce
```

2. 支持的版本

- 8.9.5-ce.0 rc latest
- 8.9.4-ce.1
- 8.8.7-ce.1
- 8.7.9-ce.1

3. 使用镜像

镜像中有三个目录用于保存 Gitlab 的数据，出于安全考虑，应该使用主机目录挂载这三个卷做持久化存储。如果容器出错，则可以启动新容器挂载这三个目录，继续提供服务，避免数据丢失。

/etc/gitlab　包含 Gitlab 的配置文件。

/var/opt/gitlab　Gitlab 使用的仓库，保存 git 的所有版本库。

/var/log/gitlab　包含 Gitlab 的日志。

直接启动 Gitlab 容器，提供服务。

```
$ docker run --name gitlab -d \
   -v /gitlab/config:/etc/gitlab \
   -v /gitlab/repo:/var/opt/gitlab \
   -v /gitlab/log:/var/log/gitlab \
   -p 2222:22 -p 80:80 -p 433:433 \
   gitlab/gitlab-ce
```

在本例中，在主机上设置三个目录，用于挂载数据卷。为了避免 Gitlab 容器占用主机的 22 端口，把容器中的 22 端口导出到 2222。在客户端中，需要配置~/.ssh/config 使用 2222 作为 SSH 登陆 Gitlab 容器的端口。

```
$ vi ~/.ssh/config
HOST gitlab.company.com
HOSTNAME gitlab.company.com
PORT 2222
```

4. 导出端口

22——用户 SSH 登录的端口。当使用 git 协议通过免登录的方式 clone 代码时，使用此端口。

80——http 端口，通过此端口管理界面。

443——https 端口，通过此端口管理界面。

12.3　如何使用容器提供编程环境

在本节中将介绍如何使用容器提供各种编程环境。

12.3.1　使用容器提供 Java 环境

Java 是一种面向对象的编程语言。Java 具有跨平台、面向对象、泛型编程的特性，广泛应用于企业级 Web 应用开发和移动应用开发。Java 不仅吸收了 C++语言的各种优点，还摒弃了 C++难以理解的多继承、指针等概念。因此，Java 语言具有功能强大和简单易用两个特征。Java 语言作为静态面向对象编程语言的代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程。

1. 镜像名

在 Docker Hub 中，Java 的官方镜像是 java，可以通过下面的命令下载镜像。

```
docker pull java
```

2. 支持的版本

在镜像中，不同版本使用不同 tag 区分。同一版本可能有不同 tag，支持的版本如下。

- 6b38-jdk、6b38、6-jdk、6、openjdk-6b38-jdk、openjdk-6b38、openjdk-6-jdk、openjdk-6；
- 6b38-jre、6-jre、openjdk-6b38-jre、openjdk-6-jre；
- 7u101-jdk、7u101、7-jdk、7、openjdk-7u101-jdk、openjdk-7u101、openjdk-7-jdk、openjdk-7；
- 7u91-jdk-alpine、7u91-alpine、7-jdk-alpine、7-alpine、openjdk-7u91-jdk-alpine、openjdk-7u91-alpine、openjdk-7-jdk-alpine、openjdk-7-alpine；
- 7u101-jre、7-jre、openjdk-7u101-jre、openjdk-7-jre；
- 7u91-jre-alpine、7-jre-alpine、openjdk-7u91-jre-alpine、openjdk-7-jre-alpine；
- 8u91-jdk、8u91、8-jdk、8、jdk、latest、openjdk-8u91-jdk、openjdk-8u91、openjdk-8-jdk、openjdk-8；
- 8u92-jdk-alpine、8u92-alpine、8-jdk-alpine、8-alpine、jdk-alpine、alpine、openjdk-8u92-jdk-alpine、openjdk-8u92-alpine、openjdk-8-jdk-alpine、openjdk-8-alpine；
- 8u91-jre、8-jre、jre、openjdk-8u91-jre、openjdk-8-jre；
- 8u92-jre-alpine、8-jre-alpine、jre-alpine、openjdk-8u92-jre-alpine、openjdk-8-jre-alpine；
- 9-b124-jdk、9-b124、9-jdk、9、openjdk-9-b124-jdk、openjdk-9-b124、openjdk-9-jdk、openjdk-9；
- 9-b124-jre、9-jre、openjdk-9-b124-jre、openjdk-9-jre。

3. 使用镜像

最直接的方式是在 java 容器中编译和运行 Java 程序。可以使用 java 镜像作为基础镜像，制作定制化的镜像。

```
FROM java:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

编译这个镜像并启动容器。

```
$ docker build -t my-java-app .
$ docker run -it --rm --name my-running-app my-java-app
```

在某些场合，不能在容器中运行 Java 程序，如与驱动相关的 Java 程序。此时，可以使用 java 容器仅编译工程，得到二进制文件。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp java:7 javac
Main.java
```

把宿主机上的工作目录当作卷挂载到容器中。在工作目录中使用 javac Main.java 命令编译工程，得到类文件 Main.class。

4. 镜像分类

java 镜像有不同的标签，可应用在不同场景。

（1）java:<version>镜像是默认的镜像。当不确定需求时，可以使用这个镜像。这个镜像在容器中挂载用户源码，编译程序，并运行程序。这类镜像是基于 buildpack-deps 的基础镜像，buildpack-deps 包含了通常开发所必须的头文件和工具。

（2）java:alpine 镜像使用 Apline Linux 作为基础镜像。Apline Linux 是目前最小的基础镜像，存储空间小于 5MB。推荐使用该镜像，因为它占用的空间最小。需要注意的是，Apline Linux 使用 musl libc 替代 glibc。一些依赖 libc 的程序在使用该镜像的容器中运行可能会失败。幸运的是，大部分程序都能够在使用该镜像的容器中运行。为了节约空间，Alpine Linux 镜像不会包含类似 git、bash 的工具。开发者可以自己制作 Dockerfile，添加需要的工具。

12.3.2　使用容器提供 Golang 环境

Golang 又称作 Go，是 Google 开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。Rob Pike、Robert Griesemer 及 Ken Thompson 于 2007 年 9 月开始设计 Golang 语言，稍后 Ian Lance Taylor、Russ Cox 加入项目中。Golang 语言是基于 Inferno 操作系统所开发的。Golang 语言于 2009 年 11 月正式宣布推出，成为开放源代码项目，并在 Linux 及 Mac OS X 平台上进行了实现，后又追加了 Windows 系统下的实现。

1. 镜像名

在 Docker Hub 中，Golang 的官方镜像是 golang，可以通过下面的命令下载镜像。

```
docker pull golang
```

2. 支持的版本

在镜像中，不同版本使用不同 tag 区分，同一版本可能有不同 tag，支持的版本如下。

- 1.5.4、1.5
- 1.5.4-onbuild、1.5-onbuild;
- 1.5.4-wheezy、1.5-wheezy;
- 1.5.4-alpine、1.5-alpine;
- 1.6.2、1.6、1、latest;
- 1.6.2-onbuild、1.6-onbuild、1-onbuild、onbuild;
- 1.6.2-wheezy、1.6-wheezy、1-wheezy、wheezy;
- 1.6.2-alpine、1.6-alpine、1-alpine、alpine;
- 1.7rc1、1.7;
- 1.7rc1-onbuild、1.7-onbuild;
- 1.7rc1-wheezy、1.7-wheezy;
- 1.7rc1-alpine、1.7-alpine。

3. 使用镜像

Golang 容器同时提供编译环境和运行环境。开发者可以使用 golang 镜像作为基础镜像，制作定制化的 Golang 镜像。在这个镜像中，可以编译运行你的 Golang 程序。

```
FROM golang:1.6-onbuild
```

与 golang:1.6 相比，golang:1.6-onbuild 的 Dockerfile 中加入了一些 ONBUILD 指令。在编译定制化镜像时，会执行 COPY . /go/src/app、RUN go get -d -v 和 RUN go install -v 指令。可以在 Dockefile 中加入 CMD ["app"]指令，作为容器启动时运行的启动程序。

编译镜像并启动容器。

```
$ docker build -t my-golang-app .
$ docker run -it --rm --name my-running-app my-golang-app
```

可以使用 golang 容器仅编译 Golang 程序。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.6 go build -v
```

这种方式把宿主机的当前目录挂载到容器/usr/src/myapp 目录上，并在容器中切换

到/usr/src/myapp 目录中。执行 go build 编译工程。编译出的二进制文件将保存在宿主机的当前目录中。

如果有 Makefile，也可以在容器中运行 make 命令。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.6 bash -c make
```

如果要交叉编译，则可以通过环境变量设置编译环境。下例将编译运行在 windows/386 上的 Golang 程序。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp -e GOOS=windows -e
GOARCH=386 golang:1.6 go build -v
```

也可以一次编译多个平台的 Golang 程序。

```
$ docker run --rm -it -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.3-cross bash
$ for GOOS in darwin linux; do
>   for GOARCH in 386 amd64; do
>     go build -v -o myapp-$GOOS-$GOARCH
>   done
> done
```

4．镜像说明

golang 镜像有不同的标签，可应用在不同场景。

（1）golang:<version>镜像是默认的镜像。当不确定需求时，可以使用这个镜像。这个镜像在容器中挂载用户源码，编译程序，并运行程序。这类镜像是基于 buildpack-deps 的基础镜像，buildpack-deps 包含了通常开发必需的头文件和工具。

（2）golang:alpine 镜像使用 Apline Linux 作为基础镜像。Apline Linux 是目前最小的基础镜像，存储空间小于 5MB。推荐使用该镜像，因为它占用的空间最小。需要注意的是，Apline Linux 使用 musl libc 替代 glibc。一些依赖 libc 的程序在使用该镜像的容器中运行可能会失败。幸运的是，大部分程序都能在使用该镜像的容器中运行。

为了节约空间，Alpine Linux 镜像不会包含类似 git、bash 的工具。可以自己制作 Dockerfile，添加需要的工具。

（3）golang:onbuild 镜像用于编译派生镜像。通常在工程目录中创建 Dockerfile，在 Dokerfile 中使用 FROM golang:onbuild 指令。这样可以编译出一个包含二进制程序的单独镜像。在实践中，仅在开发阶段使用这类镜像。因为它可以快速生成二进制文件。不推荐在长期运行的环境中使用该类镜像。因为开启 ONBUILD 指令后，对镜像会失去完全控制。

若掌握了如何使用 Docker 运行 golang 程序，推荐使用非 onbuild 镜像作为基础镜像。这样开发者就可以完全控制自己的镜像，其他人也可以通过 Dockerfile 明白该镜像在做什么。

12.4　习题

本章介绍了几种常见镜像的使用方法。接下来，通过习题和实验检验本章的学习成果。

（1）启动一个 MySQL 容器，导出容器 3306 端口到主机 13306 端口。启动容器时，通过环境变量设置数据库密码为 ghostcloud，数据库名为 ghostclouddb。使用 MySQL Client 访问该数据库。

（2）启动一个 httpd 容器，导出容器 80 端口到主机 8080 端口。通过浏览器访问 http://ip:8080。

第 13 章　建立私有镜像仓库

Docker Registry 是镜像仓库，用于管理、保存镜像。用户可以从仓库 pull 或 push 镜像。

用户既可以使用 dockerhu.com 这样的公共仓库，也可以配置私有仓库，提供镜像仓库服务。

Docker 把镜像仓库做成了一个容器，提供镜像服务。镜像名称是 registry，在 Docker Hub 上可以查到官方最新版本。为了保证容器升级后镜像可用，可以使用挂载卷，做数据持久化。在主机中创建一个目录，挂载到容器中，作为镜像存储的目录。

```
# mkdir -p /registry/public/repos
# docker run --rm -p 5000:5000 -v /registry/public/repos:/var/lib/registry
registry
```

如果要使用非加密仓库，则需要在 Docker Daemon 中添加--insecure-registry。当 Docker Daemon 认证该仓库后，就可以从镜像仓库 pull 或 push 镜像。编辑 /etc/default/docker 文件，配置 192.168.8.1 作为 registy 容器所在主机的 IP，5000 为导出端口。

```
# vi /etc/default/docker
DOCKER_OPTS="-H=unix:///var/run/docker.sock --insecure-registry=192.168.8.1:5000"
# restart docker
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon                                 commit=20f81dd
execdriver=native-0.2 graphdriver=devicemapper version=1.10.3
INFO[0000] API listen on /var/run/docker.sock
```

现在可以上传镜像到私有仓库。首先，给镜像添加 tag，标签名为私有仓库的 URL。

```
# docker tag swarm:latest 192.168.8.1:5000/swarm:latest
```

Docker Daemon 会根据 URL 找到私有仓库，上传镜像。

```
# docker push 192.168.8.1:5000/swarm
The push refers to a repository [192.168.8.1:5000/swarm] (len: 1)
a9975e2cc0a3: Pushed
```

```
90b2ffb8d338: Pushed
188de6f24f3f: Pushed
d681c900c6e3: Pushed
latest: digest: sha256:c366027d78e36f04e9bacf15682a9d97571e2f133f27a70e50-
ee88c35cfcfb54
size: 11678
```

也可以从私有仓库中下载镜像。

```
# docker pull 192.168.8.1:5000/swarm
Using default tag: latest
latest: Pulling from swarm
Digest: sha256:c366027d78e36f04e9bacf15682a9d97571e2f133f27a70e50ee88c35cfcfb54
Status: Downloaded newer image for 192.168.8.1:5000/swarm:latest
```

13.1　镜像仓库配置详解

Docker Registry 的配置文件是一个 yaml 文件，放在容器中/etc/docker/registry/config.yml。

镜像仓库启动时，从 config.yml 读取配置。用户也可以设置环境变量，覆盖 config.yml 的配置。环境变量的命名规则是 REGISTRY_VARIABLE，VARIABLE 是 config.yml 中配置的名字，_表示配置的层级。

例如，storage->filesystem->rootdirectory 的配置如下。

```
storage:
  filesystem:
    rootdirectory: /var/lib/registry
```

如果要覆盖这个值，则可以添加环境变量。

```
REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY=/new_registry
```

在生产环境中，必须使用文件配置 registry，不能使用环境变量配置 registry。因为，一旦运行了新容器，环境变量会变化。

下面的配置项列表包括了所有的配置，其中有些配置是互斥的，在设置时必须仔细阅读配置的详细信息。

有一些选项比较特别，父选项是可选的，子选项是必选的。设置这类选项的原则是：忽略父选项和所有子选项。设置父选项的同时必须设置所有的必选子选项。

```
version: 0.1
log:
level: debug
formatter: text
fields:
  service: registry
  environment: staging
```

```
    hooks:
      - type: mail
        disabled: true
        levels:
          - panic
        options:
          smtp:
            addr: mail.example.com:25
            username: mailuser
            password: password
            insecure: true
          from: sender@example.com
          to:
            - errors@example.com
    storage:
    filesystem:
      rootdirectory: /var/lib/registry
    azure:
      accountname: accountname
      accountkey: base64encodedaccountkey
      container: containername
    gcs:
      bucket: bucketname
      keyfile: /path/to/keyfile
      rootdirectory: /gcs/object/name/prefix
    s3:
      accesskey: awsaccesskey
      secretkey: awssecretkey
      region: us-west-1
      bucket: bucketname
      encrypt: true
      secure: true
      v4auth: true
      chunksize: 5242880
      rootdirectory: /s3/object/name/prefix
    rados:
      poolname: radospool
      username: radosuser
      chunksize: 4194304
    swift:
      username: username
      password: password
      authurl:   https://storage.myprovider.com/auth/v1.0   or   https://storage.
myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
      tenant: tenantname
      tenantid: tenantid
      domain: domain name for Openstack Identity v3 API
      domainid: domain id for Openstack Identity v3 API
      insecureskipverify: true
      region: fr
      container: containername
```

```
    rootdirectory: /swift/object/name/prefix
  delete:
    enabled: false
  redirect:
    disable: false
  cache:
    blobdescriptor: redis
  maintenance:
    uploadpurging:
      enabled: true
      age: 168h
      interval: 24h
      dryrun: false
    readonly:
      enabled: false
  auth:
  silly:
    realm: silly-realm
    service: silly-service
  token:
    realm: token-realm
    service: token-service
    issuer: registry-token-issuer
    rootcertbundle: /root/certs/bundle
  htpasswd:
    realm: basic-realm
    path: /path/to/htpasswd
  middleware:
  registry:
    - name: ARegistryMiddleware
      options:
        foo: bar
  repository:
    - name: ARepositoryMiddleware
      options:
        foo: bar
  storage:
    - name: cloudfront
      options:
        baseurl: https://my.cloudfronted.domain.com/
        privatekey: /path/to/pem
        keypairid: cloudfrontkeypairid
        duration: 3000
  reporting:
  bugsnag:
    apikey: bugsnagapikey
    releasestage: bugsnagreleasestage
    endpoint: bugsnagendpoint
  newrelic:
    licensekey: newreliclicensekey
    name: newrelicname
```

```
  verbose: true
http:
addr: localhost:5000
prefix: /my/nested/registry/
host: https://myregistryaddress.org:5000
secret: asecretforlocaldevelopment
tls:
  certificate: /path/to/x509/public
  key: /path/to/x509/private
  clientcas:
    - /path/to/ca.pem
    - /path/to/another/ca.pem
debug:
  addr: localhost:5001
headers:
  X-Content-Type-Options: [nosniff]
notifications:
endpoints:
  - name: alistener
    disabled: false
    url: https://my.listener.com/event
    headers: <http.Header>
    timeout: 500
    threshold: 5
    backoff: 1000
redis:
addr: localhost:6379
password: asecret
db: 0
dialtimeout: 10ms
readtimeout: 10ms
writetimeout: 10ms
pool:
  maxidle: 16
  maxactive: 64
  idletimeout: 300s
health:
storagedriver:
  enabled: true
  interval: 10s
  threshold: 3
file:
  - file: /path/to/checked/file
    interval: 10s
http:
  - uri: http://server.to.check/must/return/200
    headers:
      Authorization: [Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==]
    statuscode: 200
    timeout: 3s
    interval: 10s
```

```
    threshold: 3
tcp:
  - addr: redis-server.domain.com:6379
    timeout: 3s
    interval: 10s
    threshold: 3
proxy:
remoteurl: https://registry-1.docker.io
username: [username]
password: [password]
```

13.2　version 选项

```
version: 0.1
```

必选项，表示配置文件版本号。这个配置必须放在顶级。在解析其他配置之前，registry 会使用这个配置做一致性检查。

13.3　log 选项

```
log:
level: debug
formatter: text
fields:
  service: registry
  environment: staging
```

配置日志系统。日志打印在终端，用户可以配置日志的频率和格式。Log 选项详解见表 13.1。

表 13.1　log 选项详解

配　置　项	必　选　项	说　　明
level	no	设置日志等级。可以设置 error、warn、info 或 debug。默认值为 info
formatter	no	设置输出格式。可以设置 text、json 或 logstash。默认值为 text
fields	no	在日志中添加关键字，关键字将以 key:value 的方式打印在每条日志中。当 registry 和其他系统一起协作时，可以添加 registry 特定的关键字，区分日志

13.4　hooks 选项

```
hooks:
  - type: mail
    levels:
      - panic
    options:
```

```
    smtp:
      addr: smtp.sendhost.com:25
      username: sendername
      password: password
      insecure: true
    from: name@sendhost.com
    to:
      - name@receivehost.com
```

hooks 可以配置日志的钩子，且可以配置多个钩子，如发送邮件等。

13.5　storage 选项

storage 是必选项，用于配置存储镜像的方式。只能选择一种后端存储，如果选择多种后端存储，则镜像仓库启动时会报错。

```
storage:
  filesystem:
    rootdirectory: /var/lib/registry
  azure:
    accountname: accountname
    accountkey: base64encodedaccountkey
    container: containername
  gcs:
    bucket: bucketname
    keyfile: /path/to/keyfile
    rootdirectory: /gcs/object/name/prefix
  s3:
    accesskey: awsaccesskey
    secretkey: awssecretkey
    region: us-west-1
    regionendpoint: http://myobjects.local
    bucket: bucketname
    encrypt: true
    keyid: mykeyid
    secure: true
    v4auth: true
    chunksize: 5242880
    rootdirectory: /s3/object/name/prefix
  swift:
    username: username
    password: password
    authurl:  https://storage.myprovider.com/auth/v1.0  or  https://storage.
myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
    tenant: tenantname
    tenantid: tenantid
    domain: domain name for Openstack Identity v3 API
    domainid: domain id for Openstack Identity v3 API
    insecureskipverify: true
```

```
    region: fr
    container: containername
    rootdirectory: /swift/object/name/prefix
  oss:
    accesskeyid: accesskeyid
    accesskeysecret: accesskeysecret
    region: OSS region name
    endpoint: optional endpoints
    internal: optional internal endpoint
    bucket: OSS bucket
    encrypt: optional data encryption setting
    secure: optional ssl setting
    chunksize: optional size valye
    rootdirectory: optional root directory
  inmemory:
  delete:
    enabled: false
  cache:
    blobdescriptor: inmemory
  maintenance:
    uploadpurging:
      enabled: true
      age: 168h
      interval: 24h
      dryrun: false
  redirect:
disable: false
```

storage 选项详解见表 13.2。

表 13.2　storage 选项详解

配　置　项	说　　明
filesystem	使用本地磁盘存储镜像
azure	使用 Microsoft's Azure Blob Storage 存储镜像
gcs	使用 Google Cloud Storage 存储镜像
s3	使用 Amazon S3 存储镜像
swift	使用 Openstack Swift 存储镜像
oss	使用阿里云 OSS 存储镜像

如果在 Windows 上创建镜像仓库，不要使用 Windows 挂载卷，可以使用 S3、Azure 等对象存储。如果要使用 Windows 挂载卷，则必须保证挂载点名字不能超过 Windows 中 MAX_PATH 的值，通常为 255 个字符，否则，会遇见如下错误。

```
mkdir /XXX protocol error and your registry will not function properly.
```

13.5.1　filesystem 选项

若使用本地文件系统存储镜像，则需要配置 filesy stem 选项。在开发环境中可以使用本地存储。当数据量要求不大时，也可以使用本地存储。语法如下：

```
filesystem:
   rootdirectory: /var/lib/registry
```

使用 rootdirectory 配置路径，rootdirectory 必须是绝对路径。使用本地存储时，要确保磁盘上有可用空间。

13.5.2　azure 选项

若使用 Microsoft Azure 存储镜像，则需要配置 azure 选项。

```
azure:
   accountname: accountname
   accountkey: base64encodedaccountkey
   container: containername
```

azure 选项详解见表 13.3。

表 13.3　azure 选项详解

配　置　项	必　选　项	说　　明
accountname	yes	微软 Azure 账号
accountkey	yes	微软 Azure Key
container	yes	微软 Azure Container
realm	no	微软 Azure 存储服务的后缀域名

13.5.3　gcs 选项

若使用 Google Cloud Storage 存储镜像，则需要配置 gcs 选项。

```
gcs:
   bucket: bucketname
   keyfile: /path/to/keyfile
    rootdirectory: /gcs/object/name/prefix
```

gcs 选项详解见表 13.4。

表 13.4　gcs 选项详解

配　置　项	必　选　项	说　　明
bucket	yes	Google Cloud Storage Bucket
keyfile	no	Google Cloud Storage 提供的 Key 文件
rootdirectory	no	bucket 中的根目录，默认为空，所有数据存在 bucket 下
chunksize	no	分片上传时，分片数据大小。默认为 5242880，该值必须为 256×1024 的倍数

13.5.4　s3 选项

若使用 Amazon S3 存储镜像，则需要配置 s3 选项。

```
s3:
   accesskey: awsaccesskey
```

```
    secretkey: awssecretkey
    region: us-west-1
    regionendpoint: http://myobjects.local
    bucket: bucketname
    encrypt: true
    keyid: mykeyid
    secure: true
    v4auth: true
    chunksize: 5242880
rootdirectory: /s3/object/name/prefix
```

s3 选项详解见表 13.5。

表 13.5　s3 选项详解

配 置 项	必 选 项	说　明
accesskey	yes	AWS 提供的 Access Key
secretkey	yes	AWS 提供的 Secret Key
region	yes	AWS 的区域，如 us-east-1。详情查看 http://docs.aws.amazon.com/AWSEC2/latest/ UserGuide/ using-regions-availability-zones.html
regionendpoint	no	兼容 S3 的存储服务
bucket	yes	AWS 使用的 bucket
encrypt	no	是否在服务端加密数据，默认为 false
keyid	no	使用 KMS key ID 加密
secure	no	是否使用 ssl 传输数据，默认为 true
v4auth	no	使用 aws signature V4
chunksize	no	分片上传时，每片数据的大小。默认为 10MB。oss 最小的分片大小为 5MB。chunksize 必须大于 5MB。增大这个值，上传的速度可能会提高
rootdirectory	no	bucket 中的根目录，默认为空，所有数据存在 bucket 下
storageclass	no	storage class 类型，默认为 STANDARD。 可选值为 STANDARD 或 REDUNDANCY

13.5.5　swift 选项

若使用 OpenStack Swift 存储镜像，则需要配置 swift 选项。

```
swift:
    username: username
    password: password
    authurl: https://storage.myprovider.com/auth/v1.0 or https://storage.
myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
    tenant: tenantname
    tenantid: tenantid
    domain: domain name for Openstack Identity v3 API
    domainid: domain id for Openstack Identity v3 API
    insecureskipverify: true
    region: fr
```

```
    container: containername
rootdirectory: /swift/object/name/prefix
```

swift 选项详解见表 13.6。

表 13.6　swift 选项详解

配 置 项	必 选 项	说　明
authurl	yes	通过该 URL 获得 auth token。 如 https://storage.myprovider.com/v2.0 或 https://storage.myprovider.com/v3/auth
username	yes	Openstack 用户名
password	yes	Openstack 密码
region	no	OpenStack 区域
container	yes	Openstack 中的 Swift Container
tenant	no	Openstack tTenant 名字。tenant 和 tenantid 二选一
tenantid	no	Openstack Tenant ID。tenant 和 tenantid 二选一
domain	no	Openstack Domain 名字。domain 和 domainid 二选一
domainid	no	Openstack Domain ID。domain 和 domainid 二选一
trustid	no	Openstack Trust ID
insecureskipverify	no	设置为 true 时，跳过 TLS 认证。默认值为 false
chunksize	no	分片上传时，每片数据的大小。默认为 5MB
prefix	no	Openstack Swift 秘钥使用的前缀
secretkey	no	生成临时 URL 使用的加密秘钥
accesskey	no	生成临时 URL 使用的访问秘钥

13.5.6　oss 选项

若使用阿里云阿里云 OSS 存储镜像，则需要配置 oss 选项。

```
oss:
    accesskeyid: TXBvStdEm3tLB50x
    accesskeysecret: txa7YVT3bbqLXHj12I2IMNnZn4tora
    region: oss-cn-hangzhou
    bucket: bucket1
    endpoint: bucket1.oss-cn-hangzhou-internal.aliyuncs.com
    internal: true
    encrypt:
    secure: true
    chunksize:
    rootdirectory: test
```

oss 选项详解见表 13.7。

表 13.7　oss 选项详解

配　置　项	必　选　项	说　　明
accesskeyid	yes	阿里云分配给用户的 Access Key ID
accesskeysecret	yes	阿里云分配给用户的 Access Key Secret
region	yes	oss 的区域，如 oss-cn-beijing、oss-cn-hangzhou 等。具体可选区域参考阿里云官方文档。https://help.aliyun.com/document_detail/oss/user_guide/endpoint_region.html
endpoint	no	系统会根据 region、bucket、secure 信息生成默认访问域名。secure: false 时，使用<bucket>.<region>.aliyuncs.com；secure: true 时，使用<bucket>.<region>-internal.aliyuncs.com。也可以手动配置访问域名
internal	no	是否使用 oss 内网域名。true 时，使用<bucket>.<region>-internal.aliyuncs.com；false 时，使用<bucket>.<region>.aliyuncs.com。默认为 false
bucket	yes	使用的 bucket。使用前，必须在阿里云上创建 bucket
encrypt	no	是否在服务端加密数据，默认为 false
secure	no	是否使用 ssl 传输数据，默认为 true
chunksize	no	分片上传时，每片数据的大小。默认为 10MB。oss 最小的分片大小为 5MB，chunksize 必须大于 5MB。增大这个值，上传的速度可能会提高
rootdirectory	no	bucket 中的根目录，默认为空，所有数据存在 bucket 下

13.5.7　delete 选项

delete 选项默认为 false。如果开启，则可以根据 digest 删除镜像的 blobs 和 manifests。

```
delete:
  enabled: true
```

13.5.8　cache 选项

cache 选项用于开启缓存。目前，只能缓存镜像层的元数据，而不能缓存镜像数据。

```
cache:
blobdescriptor: inmemory
```

若使用 blobdescriptor 配置缓存，则可以设置 redis 或 inmemory。redis 使用 Redis pool 作为缓存。inmemory 使用 memory map 作为缓存。

注意　在以前的版本中，使用 layerinfo 配置缓存。目前的版本 layerinfo 已经被淘汰，只能使用 blobdescriptor。

13.5.9　maintenance 选项

maintenance 选项用于设置镜像仓库的维护模式，目前只能在维护模式中设置 uploadpurging 和 readonly。

```
maintenance:
   uploadpurging:
```

```
    enabled: true
    age: 168h
    interval: 24h
    dryrun: false
  readonly:
enabled: false
```

uploadpurging 是一个后台的清理进程，会定期从 upload 目录中删除孤立文件。uploadpurging 默认开启。

maintenance 选项详解见表 13.8。

表 13.8　maintenance 选项详解

配 置 项	必 选 项	说　明
enabled	yes	开启清理进程
age	yes	upload 目录的有效期，超过这个有效期以后，upload 目录会被删除。默认为一周（168h）
interval	yes	清理进程扫描频率，默认为一天（24h）扫描一次
dryrun	yes	如果开启，则进入检测模式。检测模式下，仅扫描孤立文件，而不删除。默认为 false

注意　age 和 interval 的格式为 45m、2h10m、168h (1 week)。

开启 readonly 模式，客户端不能 push 镜像，只能 pull 镜像。当在后端存储上开启垃圾回收进程时，需要把 registry 临时设置为只读模式。首先在配置文件中开启只读模式，并重启仓库。然后开启垃圾回收进程。当垃圾回收完成后，关闭只读模式，并重启仓库。

13.5.10　redirect 选项

redirect 选项用于配置后端存储是否使用重定向。

```
redirect:
disable: false
```

如果后端存储支持重定向，则默认开启重定向。在某些情况下，用户希望不使用重定向，要求所有数据都要经过 registry 路由。当后端存储和 registy 不在一个机房，或要求 registry 强制缓存时，则取消重定向可能更有效。设置 disable:true，关闭重定向。

13.6　auth 选项

auth 选项用于配置认证方式，是可选项。目前有三种可选认证方式：silly、token 和 htpasswd。配置时，只能配置其中一种方式。

```
auth:
  silly:
    realm: silly-realm
    service: silly-service
  token:
    realm: token-realm
    service: token-service
    issuer: registry-token-issuer
    rootcertbundle: /root/certs/bundle
  htpasswd:
    realm: basic-realm
path: /path/to/htpasswd
```

13.6.1　silly 选项

silly 选项只能用于开发测试。这种方式只检查 HTTP 请求中是否包含 Authorization 头，而不检查其中的值。如果请求中没有包含 Authorization 头，则 silly 认证会返回一个 challenge 响应，打印 realm、service 和 scope 信息。

silly 选项详解见表 13.9。

表 13.9　silly 选项详解

配　置　项	必　选　项	说　　明
reaml	yes	认证端
service	yes	服务名

13.6.2　token 选项

token 方式允许认证服务区与 registry 分离，两者使用 SSL 方式通信，这是一种非常安全的认证方式。

token 选项详解见表 13.10。

表 13.10　token 选项详解

配　置　项	必　选　项	说　　明
reaml	yes	认证端
service	yes	服务名
issuer	yes	token 的发布服务名，发布服务会把这个值插入到 token 中，这个值必须与认证服务器上的发布服务名一致。
rootcertbundle	yes	SSL 公钥的绝对路径，registry 使用这个公钥与认证服务器建立 SSL 通信

13.6.3　htpasswd 选项

htpasswd 方式使用 Apache HTPasswd File 进行认证，仅支持 bcrypt 的密码加密方式。registry 启动时，会一次性加载 htpasswd 文件。如果文件不可用，则 registry 会启

动失败，并报错。这种方式必须配置 TLS，因为认证构成会把 password 放在 HTTP 请求头中。

htppasswd 选项详解见表 13.11。

表 13.11　htpasswd 选项详解

配　置　项	必　选　项	说　　明
reaml	yes	认证端
path	yes	htpasswd 文件路径

13.7　middleware 选项

middleware 为可选项，用于设置不同模块使用的中间件。

```
middleware:
  registry:
    - name: A Registry Middleware
      options:
        foo: bar
  repository:
    - name: A Repository Middleware
      options:
        foo: bar
  storage:
    - name: cloudfront
      options:
        baseurl: https://my.cloudfronted.domain.com/
        privatekey: /path/to/pem
        keypairid: cloudfrontkeypairid
        duration: 3000s
```

registry 中间件必须实现 distribution.Namespace 接口；repository 中间件必须实现 distribution.Repository 接口；storage 中间件必须实现 driver.StorageDriver 接口。

目前，镜像仓库仅支持 cloudfont 中间件，它是一个存储中间件。

通过设置 cloudfont 选项，镜像仓库可以使用 AWS CloudFront 提供的 CND 服务。该服务帮助开发人员和企业在低延迟、高速数据传输、无最低使用承诺的环境下向最终用户轻松发布内容。

cloudfront 选项详解见表 13.12。

表 13.12　cloudfront 选项详解

配 置 项	必 选 项	说　明
baseurl	yes	Cloudfront 的服务器地址。格式为 SCHEME://HOST[/PATH]
privatekey	yes	AWS Cloudfront 私钥
keypairid	yes	AWS 提供的密钥对 ID
duration	no	设置发布周期，默认为 20m。格式为整数+时间单位，中间没有空格，如 3000s。可用的时间单位有 ns、us、ms、s、m 和 h。

13.8　reporting 选项

reporting 为可选项，用于设置报告工具。目前仅支持 Bugsnag 和 New Relic，可以同时使用这两个工具。

```
reporting:
  bugsnag:
    apikey: bugsnagapikey
    releasestage: bugsnagreleasestage
    endpoint: bugsnagendpoint
  newrelic:
    licensekey: newreliclicensekey
    name: newrelicname
    verbose: true
```

13.8.1　bugsnag 选项

Bugsnag 是一个自动监测程序崩溃的平台。程序崩溃时，bugsnag 可以实时监测这个过程，并进行端到端的跟踪，以及提供各种关联分析和报告，方便开发者找出问题根源。

可以使用 bugsnag 提供的服务记录镜像仓库的出错和诊断信息。

bugsnag 选项详解见表 13.13。

表 13.13　bugsnag 选项详解

配 置 项	必 选 项	说　明
apikey	yes	Bugsnag 提供的 API Key
releasestage	no	记录镜像仓库部署在什么环境下，如 product、staging 或 development
endpoint	no	设置 Bugsnag endpoint

13.8.2　newrelic 选项

New Relic 是一个用户程序实时监控平台，可以远程监控应用程序，并生成性能分析报告。

可以使用 newrelic 提供的服务记录镜像仓库的出错和诊断信息。

newrelic 选项详解见表 13.14。

表 13.14　newrelic 选项详解

配　置　项	必　选　项	说　　明
licensekey	yes	New Relic 提供的 License Key
name	no	New Relic 的应用名
verbose	no	把 New Relic 的信息打印在终端上

13.9　http 选项

配置 registry 中的 HTTP server 选项。

```
http:
  addr: localhost:5000
  net: tcp
  prefix: /my/nested/registry/
  host: https://myregistryaddress.org:5000
  secret: asecretforlocaldevelopment
  tls:
    certificate: /path/to/x509/public
    key: /path/to/x509/private
    clientcas:
      - /path/to/ca.pem
      - /path/to/another/ca.pem
  debug:
    addr: localhost:5001
  headers:
X-Content-Type-Options: [nosniff]
```

http 选项详解见表 13.15。

表 13.15　http 选项详解

配　置　项	必　选　项	说　　明
addr	yes	仓库的地址。格式和 net 设置有关。如果是 tcp，则使用 HOST:PORT 格式。如果是 unix，则使用 FILE
net	no	网络类型。UNIX 或 TCP，默认为 TCP
prefix	no	如果 HTTP server 没有运行在根目录，则使用这个值作为前缀。根目录是 v2 之前的部分。prefiex 前后都必须有/，如/path/
host	no	设置镜像仓库的访问地址
secret	yes	随机字符串，用于签名客户端上的状态，防篡改。在生产环境中，应使用专用工具生成随机串。如果配置集群 registry，则这个值在所有 registry 上必须相同

13.9.1　tls 选项

tls 为可选项，用于配置 TLS 相关信息。如果运行 registry 的主机上同时运行了 Nginx 或 Apache，则可以配置 TLS，并把它连接代理到 registry 上。

tls 选项详解见表 13.16。

表 13.16　tls 选项详解

配 置 项	必 选 项	说　明
certificate	yes	x509 证书文件的绝对路径
key	yes	x509 私钥文件的绝对路径
clientcas	no	x509 CA 文件的绝对路径列表

13.9.2　debug 选项

debug 为可选项，用于配置 debug 服务器。debug 服务器保存镜像仓库的调试信息。在生产环境中必须保证 debug 服务器的安全，为此只需设置 addr，格式为 HOST:PORT。

13.9.3　headers 选项

headers 为可选项，用于设置 HTTP 服务器的响应头，也可以设置安全相关的头，如 Strict-Transport-Security。

header 内为每个头设置一个值，参数名为 head 名，参数值为列表。

推荐设置 X-Content-Type-Options: [nosniff]，如果 registry 生成重定向，则浏览器不会把内容翻译成 HTML。

13.10　notifications 选项

notifications 选项用于配置镜像仓库的通知服务。当镜像仓库中发生某些事件时，可以对外发送通知信息。

notifications 为可选项，目前只有 endpoints 选项。

```
notifications:
  endpoints:
    - name: alistener
      disabled: false
      url: https://my.listener.com/event
      headers: <http.Header>
      timeout: 500
      threshold: 5
backoff: 1000
```

endpoints 是一个列表，该列表包含了多个服务，每个服务都可以接收镜像仓库发出的事件通知。

endpoints 选项详解见表 13.17。

表 13.17　endpoints 选项详解

配　置　项	必　选　项	说　　明
name	yes	设置服务名
disabled	no	设置为 true 时，仓库不会发送事件通知到该服务
url	yes	发送事件通知到该 URL
headers	yes	设置 http header，值为列表
timeout	yes	设置 http 超时时间，时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
threshold	yes	设置阈值，阈值为整数
backoff	yes	设置回退时间，时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns。

13.11　redis 选项

Redis 是一个高性能的 Key-Value 数据库。镜像仓库可以使用 redis 服务器缓存不会变化的 blobs 信息。

```
redis:
  addr: localhost:6379
  password: asecret
  db: 0
  dialtimeout: 10ms
  readtimeout: 10ms
  writetimeout: 10ms
  pool:
    maxidle: 16
    maxactive: 64
idletimeout: 300s
```

redis 为可选项。

redis 选项详解见表 13.18。

表 13.18　redis 选项详解

配　置　项	必　选　项	说　　明
addr	yes	设置 redis instance 地址
password	no	设置 redis instance 密码
db	no	选择 DB
dialtimeout	no	设置 redis 连接超时时间
readtimeout	no	设置 redis 读超时时间
writetimeout	no	设置 redis 写超时时间

redis 中的 pool 选项的详细参数详见表 13.19。

表 13.19　pool 选项详解

配　置　项	必　选　项	说　　明
maxidle	no	设置 idle 最大连接数
maxactive	no	设置最大连接数
idletimeout	no	设置限制连接的超时时间

13.12　health 选项

health 为可选项，用于监控镜像仓库各个模块的可用性。可以监控镜像的后端存储、本地文件、HTTP 连接和 TCP 连接。如果设置了 debug 服务器，则监控结果将上传到 debug 服务器中。

```
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
  file:
    - file: /path/to/checked/file
      interval: 10s
  http:
    - uri: http://server.to.check/must/return/200
      headers:
        Authorization: [Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==]
      statuscode: 200
      timeout: 3s
      interval: 10s
      threshold: 3
  tcp:
    - addr: redis-server.domain.com:6379
      timeout: 3s
      interval: 10s
threshold: 3
```

13.12.1　storagedriver 选项

storagedriver 选项可以监控镜像仓库中后端存储的运行情况。将 storagedriver 中的 enable 选项设置为 yes，可开启对后端存储的监控。

storagedriver 的详细参数见表 13.20。

表 13.20　storagedriver 选项详解

配 置 项	必 选 项	说　明
enabled	yes	设置为 true 时，监控后端存储
interval	no	设置扫描周期，默认为 10s。时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
threshold	no	设置连接失败之前的重试次数

13.12.2　file 选项

file 选项是一个文件列表，每条记录标识了一个文件路径。如果该文件存在，则检查会失败。在镜像仓库回滚时，可以创建一个文件。通过 file 检测该问题是否存在，从而判断回滚操作是否完成。

file 选项详解见表 13.21。

表 13.21　file 选项详解

配 置 项	必 选 项	说　明
file	yes	设置需要监控的文件路径，可以设置多个文件
interval	no	设置扫描周期，默认为 10s 时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns

13.12.3　http 选项

http 选项是一个 URI 列表，每条记录标识了一个 URI 地址。在 URI 记录中，可以监控请求头或返回码。一旦镜像仓库没有完成请求头或返回码错误，则检查会失败。

http 选项详解见表 13.22 所示。

表 13.22　http 选项详解

配 置 项	必 选 项	说　明
uri	yes	设置监控的 URI，可以设置多个 URI
headers	no	设置 http header，值为列表
statuscode	no	设置 http 连接的返回码，默认为 200
timeout	no	设置 http 超时时间，时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
interval	no	设置扫描周期，默认为 10s。时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
threshold	no	设置连接失败之前的重试次数

13.12.4　tcp 选项

tcp 选项是一个 TCP 地址列表，每条记录标识了一个需要连接的 TCP 地址和端口号。镜像仓库会尝试连接这些 TCP 地址，一旦连接失败，则检查会失败。

tcp 选项详解见表 13.23。

表 13.23　tcp 选项详解

配　置　项	必　选　项	说　　明
addr	yes	设置监控的 TCP 地址和端口
timeout	no	设置 TCP 超时时间，时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
interval	no	设置扫描周期，默认为 10s。时间为整数+单位。整数必须是正数。单位有 ns、us、ms、s、m 和 h。如果忽略单位，则时间为 ns
threshold	no	设置连接失败之前的重试次数

13.13　proxy 选项

proxy 为可选项，用于设置仓库作为 mirror 镜像仓库。当设置镜像仓库为 mirror 时，镜像仓库会从 dockerhub.com 同步镜像。

```
proxy:
  remoteurl: https://registry-1.docker.io
  username: [username]
  password: [password]
```

proxy 选项详解见表 13.24。

表 13.24　proxy 选项详解

配　置　项	必　选　项	说　　明
remoteurl	yes	Docker Hub 的官方 URL
username	no	Docker Hub 上的用户名
password	no	Docker Hub 上的密码

13.14　镜像仓库配置实例

镜像仓库的配置复杂、烦琐。本节列出一些镜像仓库的典型配置。

13.14.1　启动容器数据持久化

镜像仓库以容器的方式运行，并提供镜像下载上传服务。容器本身不能保证数据持久化，如果容器升级，则之前的镜像文件就不能保存下来。镜像文件会占用很大的存储空间，容器中的存储空间是有限制的，如果超过限制，则镜像文件也不能保存下来。

在实际操作中，当需要在运行容器时，则使用挂载卷的方式保证数据持久化。

（1）在宿主机上新建文件夹 datastore，保存镜像数据。

```
# mkdir /registry/datastore
```

（2）在宿主机上新建镜像仓库配置文件 config.yml，在该文件中加入镜像仓库的配置。

```
# vi /registry/conf/config.yml
version: 0.1
log:
  level: debug
storage:
    filesystem:
        rootdirectory: /var/lib/registry
```

（3）启动镜像仓库容器。

- 使用-d，让容器运行在后台。
- 使用-v 挂载主机卷。
- 使用-p 导出 http 端口，默认情况下，镜像仓库使用 5000 端口，提供 HTTP 服务。
- 使用--restart=always，保证容器随宿主机一起重启。

```
# docker run -d -v /registry/datastore:/var/lib/registry -v /registry/conf/config.yml:
/etc/docker/registry/config.yml:ro -p 80:5000 --restart-always registry:2.3.0
```

13.14.2　使用文件系统保存镜像

镜像保存在/var/lib/registry 目录下，运行容器时，可以把容器中的该目录挂载到宿主机上。镜像仓库绑定在容器中的 5000 端口，启动容器时，需要把 5000 端口导出。配置 debug 服务器，保存镜像仓库的调试信息。

```
version: 0.1
log:
  level: debug
storage:
    filesystem:
        rootdirectory: /var/lib/registry
http:
    addr: localhost:5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

13.14.3　使用对象存储保存镜像

镜像仓库可以使用阿里云 OSS 存储镜像。

```
version: 0.1
log:
```

```
    level: debug
    formatter: text
storage:
oss:
      accesskeyid: TXBvKtc4b9tLP50c
      accesskeysecret: gxq7YVT3bmlLPHjpoI2ILNnEn4torQ
      region: oss-cn-hangzhou
      internal: true
      bucket: registry
      secure: false
      chunksize: 10485760
      rootdirectory: datastore
http:
    addr: :5000
    net: tcp
    secret: asecretforlocaldevelopment
    headers:
       X-Content-Type-Options: [nosniff]
```

需要在阿里云创建一个 Bucket，作为镜像文件的后端存储，本例中为 registry。设置 rootdirectory:datastore，镜像仓库会自动在 Bucket 下创建 datastore 目录，作为镜像的根目录。从阿里云的管理平台上获得用户的 Access Key ID 和 Access Key Secret，作为认证信息，如图 13.1 所示。

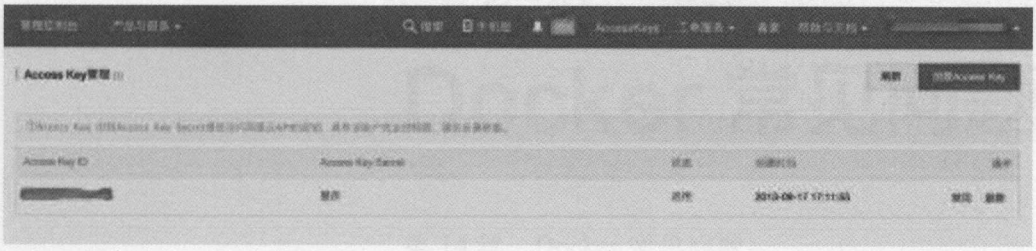


图 13.1　阿里云中查看 Access Key ID 和 Access Key Secret

13.14.4　通过中间件使用 CDN 服务

在镜像仓库中可以配置存储中间件。通过中间件，镜像仓库可以通过 CDN 服务下发镜像，提高镜像下载速度。目前，镜像仓库仅支持 Amazon Cloudfront。

```
middleware:
    storage:
       - name: cloudfront
         disabled: false
         options:
           baseurl: http://d111111abcdef8.cloudfront.net
           privatekey: /path/to/asecret.pem
           keypairid: asecret
duration: 60
```

13.15　习题

本章详细介绍了私有镜像仓库的搭建过程。接下来，通过习题和实验检验本章的学习成果。

（1）修改镜像仓库日志登记为 info。

（2）在主机上新建/docker/registry 目录，作为镜像和容器保存的位置。启动 registry 容器，挂载主机/docker/registry 目录。

第 4 篇

Docker 常见问题

第 14 章　Docker 常见问题

第 14 章　Docker 常见问题

精灵云是国内最早的从事容器研究的公司之一。该公司目前推出了精灵云容器管理平台（www.ghostcloud.cn）。该平台为用户提供了主机托管服务，为用户管理自己主机上的容器。很多用户对容器技术非常感兴趣，但是在使用过程中，遇到了各种各样的问题和困难。精灵云平台每天都在为这些用户答疑解惑。同时，本书也搜集和总结了一些常见问题，在这里逐一罗列出来，希望能够帮助广大读者。

14.1　Docker 基础问题

本节列出了一些 Docker 相关的基础问题。

14.1.1　什么是虚拟化技术

简单地讲，虚拟化是一种为了实现分时复用的技术。为了在一台物理机上跑更多的进程，并保证进程之间互不影响。虚拟化技术技术做个比喻来说，就类似用户有套房子，然后把里面隔成很多小的单间用于出租，因此通常可以提高房屋利用率，获得更好的收益。从技术角度讲，虚拟化就是在一台主机上运行多个相互隔离的实例，这包含两层意思，一是相互隔离，即彼此之间没有影响；二是实例，既可以是一个完整的操作系统，如 Windows、Linux 系统，也可以是可执行的程序。

14.1.2　虚拟化有哪些分类

硬件级虚拟化是在硬件层之上进行虚拟，通常是虚拟出独立的操作系统，如 VMWare、Xen、KVM 等。这种虚拟化技术使用 hypervisor 管理虚拟机。

操作系统级虚拟化是指在操作系统上进行虚拟，通常是虚拟出独立的容器，也称为容器虚拟化，如 Docker、CoreOS 等。通常，容器虚拟化比 hyper-v 虚拟化更轻量，更节省资源。

14.1.3　Docker 目前支持哪些操作系统

Docker 最初依赖于 Linux 内核的一些功能，所以最开始只能在 Linux 系统下运行，常用发行版本中 Cent OS、Ubuntu、SuSE、Federa 都支持 Docker。2015 年微软在 Windows Server 2016 下实现了 Docker，因此也能够在 Windows 下运行，只不过技术成熟度还不够，但是日后肯定会日趋完善。除此之外，Docker 提供了 Docker Machine，让其他操作系统也能运行 Docker。只不过，这并不是原生态运行，而是在用户自己的操作系统上运行一个 VM，然后通过定制的 Docker Client 远程操作，是一种变相的运行，因此只能用于开发测试环境。

14.1.4　哪种系统最适合运行 Docker

目前跑在 Ubuntu 下最适合，因为 Ubuntu 内核支持 AUFS，其他系统只能用 Device-mapper，AUFS 要比 Device mapper 快。

14.1.5　Docker 有什么好处

（1）标准化应用发布。Docker 容器包含了运行环境和可执行程序，可以跨平台和主机使用。

（2）节约时间。快速部署和启动，VM 启动一般是分钟级，Docker 容器启动是秒级。

（3）方便构建基于 SOA 架构或微服务架构的系统。通过服务编排，可更好地松耦合。

（4）节约成本。以前一个虚拟机至少需要几个 G 的磁盘空间，Docker 容器可以减少到 MB 级。

（5）方便持续集成。通过与代码进行关联使持续集成非常方便。

（6）可以作为集群系统的轻量主机或节点。在 IaaS 平台上已经出现了 CaaS，通过容器替代原来的主机。

14.1.6　容器化技术是什么时候出现的

2000 年出现的 FreeBSD Jail，是第一个功能完整的操作系统级虚拟化技术。真正的容器化技术出现到现在已经过去了 16 年时间，并不是几年的时间。

14.1.7　Docker 和虚拟机有什么区别

VM 主要用于硬件资源的虚拟，Docker 主要是进程级别的虚拟化，是对操作系统的虚拟。因此，如果用户只是关注业务本身，就没有必要去虚拟硬件资源，Docker 的运行速度及部署速度也会快很多。Docker 运行更快，资源占用更少，管理更方便。

14.1.8　使用 Docker 容器需要什么基础知识

使用 Docker 需要了解的一些预备知识，主要包括 Linux 和公有云的使用。因为 Docker 主要是运行在 Linux 下的，而且它的基础镜像也是基于 Linux，如果用户了解 Linux 将会避免很多障碍。公有云是日后的趋势，也是 Docker 为什么流行的因素。Linux 是开源软件的鼻祖，Docker 目前基本上都依赖 Linux 的内核，所以首先需要了解 Linux。公有云主机，其实就是将主机托管到公网上，用户不需要去关心如何管理主机，把这些事情交给专业的厂商即可。

14.1.9　如何学习 Docker

（1）如果没有云计算的基本知识，以及内核的基本知识，那么学习并理解 Docker 会稍吃力。作为容器，Docker 容器的优势在哪，不足在哪，最好了解容器的实现是怎样的（简单了解）；拥有镜像管理，Docker 又该如何体现软件开发、集成、部署、发布、再迭代的软件生命周期管理优势。

（2）关于学习资源，起码的硬件设施总是要有的。Docker 及其生态的发展很快，不使用纯理论肯定收效甚微。另外，资源的来源还包括 Docker 官方、各大电子媒体平台、技术论坛、开源社区等，往往资深人士的观点能点破自己的困惑，或者让自己知道哪方面的认识还很欠缺，以及让自己少走很多的弯路。

（3）应该认同 Docker 的设计价值，以及 Docker 的未来潜力，有依据地批判 Docker 并主动地思考，也是深切关注的表现。

（4）如果需要把 Docker 当作软件生命周期管理工具，那么用好 Docker 最为重要，API 及命令的理解与使用是必需的。如果专注系统设计方面，则除 Docker 以上的知识与经验外，应有 Docker 源码的理解与学习，才能使 Docker 水平提高一个层次。

总结起来学习 Docker 需要了解以下相关知识点：

- 云计算概念相关（RESTFUL API、微服务、OpenStack）；
- Linux 系统管理（软件包管理、用户管理、进程管理等）；
- Linux 内核相关（CGroup、Namespace 等）；
- Linux 文件系统和存储相关（AUFS、Btrfs、Devicemapper 等）；
- Linux 网络（网桥、veth、iptables 等）；
- Linux 安全相关（Appmor、Selinux 等）；
- Linux 进程管理（Supervisord、Systemd 等）；
- Linux 容器技术（LXC 等）；
- 开发语言（Python、Golang、Shell 等）。

14.2　Docker 高级问题

Docker 的应用越来越广泛，各个行业都有使用 Docker 的需求。但是，每个应用场景都有关注的重点，如网络、存储、安全、性能等。Docker 本身有丰富的选项和配置，可以满足不同的应用场景。了解 Docker 的一些选项，能够帮助开发者更好地设计出在特定场景下 Docker 的解决方案。

14.2.1　Docker 是否安全

Docker 本身是共享操作系统的进程，不存在不安全一说。如果 Docker 不安全，那么所有的 Linux 程序都是不安全的，而目前全球基本上 90%以上的网站都是运行在 Linux 上的。Docker 本身是容器技术的一种，所谓容器就像一个盒子，开发者只需暴露需要暴露的端口，如一个网站就只暴露 80 端口。而传统的服务器和云主机，基本是开发了所有的端口，或者是大多数端口，这种暴露其实是很危险的，因此 Docker 反而会让系统更安全可靠。

14.2.2　如何修改已经运行的容器

对于运行中的容器可以通过以下命令在容器中，启动一个 shell 进程。

```
# docker exec -it <container_id> bash
```

14.2.3　容器有哪些网络模式

1. None

在该模式下容器没有对外网络，本地机只有一个回路地址。

2. Container

在该模式下，与另一个容器共享网络。

3. Host

在该模式下，与主机共享网络。

4. Bridge

该模式为 Docker 默认的网络模式，在这种模式下，Docker 容器与外部的通信都是通过 iptable 实现的。

5. Overlay

该模式为 Docker 目前原生的跨主机多子网模型，主要是通过 vxlan 技术实现。

14.2.4　容器如何进行持久化

容器在退出后并不会更改镜像。因此，如果希望保存容器中的数据，就需要通过 commit 保存成镜像。另外，用户可以使用 Volume 实现数据持久化存储，保存在容器中产生或使用的文件。容器可以把数据写在 Volume 上，Volume 可以在不同的容器之间共享和重用数据，而且容器数据的备份、恢复和迁移都可以通过 Volume 实现。

14.2.5　为什么进入容器，但退出后容器就停止了

容器停止就说明容器内的主进程进程结束了。在启动容器时，如果用户是将容器放在后台运行的，并且使用 service xxx start 命令作为容器启动命令，就会产生这个问题，这个命令执行成功后就会退出，紧接着容器也会退出。因为容器的生存周期是直接和启动容器的命令生命周期一致的，一旦主进程退出，整个容器就结束了。

14.2.6　容器停止了，如何分析原因

可以用--restart 参数指定当容器退出后的行为。当容器在重启时，docker ps 可以看到处于 Up 或 Restarting，也可以在 docker events 中看到相关信息。容器的退出状态就是执行命令的错误码，Linux 下一般用 0 表示正常，其他表示错误。

同时，可以通过 docker logs 查看容器退出前的日志。

14.2.7　Link 容器是什么意思

Docker 中的容器原则上是只运行一个程序。在使用 Docker 提供服务时，会遇到需要多个程序的情况。例如，使用容器提供 LAMP 服务，需要 Apache 的容器和 MySQL 的容器一起工作。Docker 提供了 Link 的方式解决这种问题。

容器的连接（linking）系统是除端口映射外另一种和容器中应用交互的方式。该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

14.2.8　容器环境变量有什么用途

在 Docker 中，每个程序都是放在容器中运行的，同时，程序的配置文件也放在容器中。容器一旦被删除，容器内的配置文件也消失了。有些程序的配置项非常多，运维人员会花很多时间为特定场景设计出自己的配置。如果容器消失，这些配置也付之东流。

程序配置持久化的问题可以通过挂载卷的方式解决。此时，需要在宿主机上为容器建立一个文件夹，把定制化的配置文件放在该文件夹中。启动容器时，把这个文件夹挂载到容器中，替换容器中的默认配置。这个方法也有缺点，就是当需要迁移容器到新的宿主机时，同时也需要迁移配置文件。

为了解决这个问题，Docker 设计出通过环境变量修改配置的方法。把程序和配置分离开来，方便运维人员配置程序和迁移。在 MySQL 容器中，用户可以在启动容器时，通过环境变量 MYSQL_ROOT_PASSWORD 设置数据库的 root 密码，通过 MYSQL_DATABASE 设置数据库名字。使用环境变量配置容器参数，是 Docker 推荐的方法。

14.2.9　容器中 CPU、磁盘 IO、网络损耗大吗

Docker 容器中 CPU 的损耗是没有的。

Docker 容器中磁盘 IO 的损耗小于 5%，基本可以认为是没有耗损。

若对 bridge 和 host 方式做过测试，其中 host 模式是用主机的网络，基本没有性能损失；bridge 在非满载下，没什么性能损失，在满载负荷下，桥接存在 15%左右的性能损失。举个简单立在来说，如果不用 Docker，可能支持 10000 个并发，用了 Docker 则可以支持 8500 个并发，相比于 Docker 带来的便利这是可以接受的，而且系统也不应处于满载运行。

14.3　镜像相关

镜像是 Docker 能够普及的一项重要功能。以镜像的方式提交产品，加快了产品的开发、测试、部署过程，帮助运维人员提高工作效率。随着镜像的出现产生了一种新的软件部署模式，一条命令 5 分钟之内就能配置、部署一项服务，为用户节约了软件部署时间。制作一份镜像后，可以在各个平台部署服务，也可以把容器提交为镜像后，迁移到不同平台。镜像解决了用户在异构环境下部署服务和跨平台迁移服务的问题。使用镜像才能充分体现 Docker 技术的优越性。

14.3.1　什么是 Dockerfile

Dockerfile 和 Makefile 类似，是一个镜像的编译脚本，用于生成镜像。用户可以通过 Dockerfile 描述构建镜像的步骤，并自动构建一个容器。

14.3.2　Dockerfile 书写的最佳实践是什么

（1）通过 Dockerfile 所构建的镜像应该越精简越好。

（2）尽量不要安装非必要的软件包。

（3）一个容器只运行一个单独的实例，将具有耦合度的应用分别安装到不同的容器里。

（4）慎重引入新的数据层。

（5）将准备安装的软件包安照字母顺序排列，这样可以回避重复安装软件包的情况，同时也有助于进行软件更新。通过添加“\”分割命令，可以增强代码的可读性。

（6）用官方提供的镜像版本作为基础镜像，减小镜像的体积。

（7）将多条 RUN 指令使用“\”连接起来，这样更易于理解，可以方便维护。

（8）为镜像定义一个比较通用的端口，如一个用来提供 Apache Web 服务的镜像，最好暴露 80 端口。

（9）Dockerfile 开头的几句指令应该固定下来，不要每次都随意更改，这样可以利用缓存。

（10）通过-t 标记构建镜像，有助于用户管理每个创建的镜像。

（11）不要在 Dockerfile 中映射公有端口。

（12）使用 CMD 和 ENTRYPOINT 时，一定要用数组语法，而且 CMD 和 ENTRYPOINT 结合使用更好。

（13）不要开机初始化。

（14）push 之前，首先在本地构建一下，确保在本地构建的镜像在任何地方都可以正常运行。不要在构建中升级版本，如果更新时试图修改 init 或改变容器的内容，则更新可能会失败，还可能会产生不一致的镜像。

（15）FROM 命令应该包含基础镜像的完整仓库名和标签。

（16）使用指令组合，如 apt-get update 应该与 apt-get install 组合。

14.3.3　容器运行中 Entrypoint 和 CMD 的区别

（1）Dockerfile 里面写了 Entrypoint，那么 docker run 时无须指定运行的命令，如果指定了命令也是作为它的参数。

（2）如果 Dockerfile 里同时有 CMD，且是可执行的命令，则它在 ENTRYPOINT 之前运行；如果不可执行，则作为 Entrypoint 命令的参数追加。

14.3.4　Docker 中容器镜像的区别

镜像是文件系统和运行环境等配置信息，镜像需要在容器中运行。容器启动时，会加载镜像的文件系统及相关配置。

1. Docker 镜像

假设 Linux 内核是第 0 层，那么无论如何运行 Docker，它都是运行于内核层之上的。这个 Docker 镜像是一个只读的镜像，位于第 1 层，它不能被修改或不能保存状态。一个 Docker 镜像可以构建于另一个 Docker 镜像之上，这种层叠关系可以是多层的。第 1 层的镜像层称为基础镜像（Base Image），其他层的镜像（除了最顶层）称为父层

镜像（Parent Image）。这些镜像继承了它们的父层镜像的所有属性和设置，并在 Dockerfile 中添加了自己的配置。Docker 镜像通过镜像 ID 进行识别。镜像 ID 是一个 64 字符的十六进制的字符串。但是当运行镜像时，通常不会使用镜像 ID 引用镜像，而是使用镜像名引用。要列出本地所有有效的镜像，可以使用命令：

```
# docker images
```

镜像可以发布为不同的版本，这种机制称为标签（Tag）。

2. Docker 容器

Docker 容器可以使用以下命令创建：

```
# docker run imagename
```

它会在所有的镜像层之上增加一个可写层。这个可写层有运行在 CPU 上的进程，而且有两个不同的状态：运行态（Running）和退出态（Exited）。这就是 Docker 容器的特色。当使用 docker run 启动容器，Docker 容器就进入运行态，当停止 Docker 容器时，它就进入退出态。当有一个正在运行的 Docker 容器时，从运行态到停止态，它所做的一切变更都会永久地写到容器的文件系统中。要切记，对容器的变更是写入到容器的文件系统的，而不是写入到 Docker 镜像中的。可以用同一个镜像启动多个 Docker 容器，这些容器启动后都是活动的，彼此还是相互隔离的。对其中一个容器所做的变更只会局限于那个容器本身。如果对容器的底层镜像进行修改，那么当前正在运行的容器是不受影响的，即不会发生自动更新现象。如果要更新容器到其镜像的新版本，那么必须当心，确保是以正确的方式构建了数据结构，否则可能导致损失容器中所有数据。Docker 使用 64 个十六进制字符来定义容器 ID，它是容器的唯一标识符。容器之间的交互是依靠容器 ID 识别的，由于容器 ID 的字符太长，因此通常只需键入容器 ID 的前面的 4 个字符即可。当然，还可以使用容器名，但显然用 4 字符的容器 ID 更为简便。

14.3.5　Docker 的镜像仓库有哪些

由于访问国外的 Docker Hub 速度非常缓慢，经常会 pull 不下来镜像，所以推荐使用由 Ghostcloud 提供的 Docker 仓库。如果通过 Ghostcloud 安装 Docker，默认可以使用 hub.ghostclou.cn 的仓库，否则需要修改一下本机的配置，即将 --insecure-registry=hub.ghostcloud.cn 添加到在/etc/default/docker 的 DOCKER_OPTS 中。

14.3.6　如何拥有私有仓库

私有镜像仓库可以保证用户应用的安全性，但是 Docker 官方提供的私有仓库默认都需要付费。对于 Ghostcloud 用户来说，凡是接入到 Ghostcloud 的服务器都可以免费在自己的私有主机创建私有仓库。

14.4　Docker 三剑客

Github 上有非常多的第三方开源项目都是围绕 Docker 展开的，如 Kubernetes、Mesos 等。这项开源项目主要涉及容器管理、集群管理等方面。Docker 公司目前也开始发起了一些容器管理和集群管理方面的开源项目。其中，最重要的就是 Docker Machine、Docker Compose 和 Docker Swarm，这三个项目并称为 Docker 三剑客。

14.4.1　什么是 Docker Machine

Docker Machine 是 Docker 官方编排项目之一，是一个简化 Docker 安装的命令行工具，可以帮助用户构建拥有 Docker 运行环境的虚拟机，并能够远程管理虚拟机及其里面的容器。

14.4.2　什么是 Docker Compose

Docker Compose 是 Docker 的一种编排服务，是一个用于在 Docker 上定义并运行复杂应用的工具，可以让用户在集群中部署分布式应用。通过 Compose，用户可以很容易地用一个配置文件定义一个多容器的应用，然后使用一条指令安装这个应用的所有依赖，完成构建。Docker Compose 解决了容器与容器之间如何管理编排的问题，适合开发和测试环境。

14.4.3　什么是 Docker Swarm

Swarm 是 Docker 公司在 2014 年 12 月初发布的一套较为简单的工具，用来管理 Docker 集群，它将一群 Docker 宿主机变成一个单一的、虚拟的主机，使用 Swarm 操作集群，会使用户感觉就像是在一台主机上进行操作。Swarm 使用标准的 Docker API 接口作为其前端访问入口，换言之，各种形式的 Docker Client 都可以直接与 Swarm 通信。

14.5　习题

本章列出了在 Docker 使用过程中的一些常见问题。接下来，通过习题和实验检验本章的学习成果。

（1）启动 Ubuntu:14.04 容器，使用 docker top 监控容器运行状态。

（2）安装 docker-compose，并使用 docker-compose 启动容器。

博文视点精品图书展台

专业典藏

移动开发

大数据 · 云计算 · 物联网

数据库

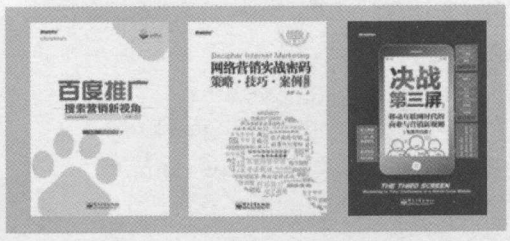
Web开发

程序设计

软件工程

办公精品

网络营销



微信公众号：ghost_cloud

目前市场上关于 Docker 的书籍偏理论化，没有专门介绍 Docker 配置、镜像制作、容器镜像如何存储的书籍。对于测试人员、运维人员和初级开发人员，无从下手。

Docker 有丰富的配置选项，用户应该根据自己的应用场景，设置合适的参数。让 Docker 运行得更稳定、安全和可监控。镜像提供了成千上万的应用，让用户可以秒级部署服务，是 Docker 能够迅速普及开来的推动力。

在集群环境部署 Docker，需要选择合适的容器和镜像存储方式，保证容器的启动速度和数据的安全性。针对这部分读者的需求，我们撰写了一本关于 Docker 的详细指南。本书从应用和实践出发，介绍了如何在用户的环境中使用 Docker，解决读者实际工作中的问题。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Docker 无疑是近年来最火热的开源技术，其掀起的容器热潮已席卷互联网技术界，越来越多的技术人员开始学习使用 Docker。目前市面上 Docker 相关书籍更多的是从源码和实现上对 Docker 进行剖析，面对初学者的书籍比较少，由精灵云主编的这本 Docker 书籍填补了这一空白。从 Docker 的安装到最佳实践，该书深入浅出地向读者展现了 Docker 的方方面面，其中很多知识点是精灵云从 Docker 领域的长期深耕实践中总结而来，对 Docker 的使用者有很强借鉴意义。我建议 Docker 的初学者、互联网公司的运维人员阅读此书，并通过章节后的习题巩固知识。该书的习题结构是一大特色，让人想起计算机领域的另一本经典书籍《深入理解计算机原理》。我有理由相信，本书也将成为 Docker 领域的一本经典书籍！

—— 张鑫，《系统虚拟化》主要作者，ZStack 创始人

精灵云作为容器技术行业的又一颗新星，实现了目前国内规模最大的自研容器调度和管理平台，积累了大量 Docker 应用和运营的一线经验，其中有许多值得我们了解和学习的独到之处。本书由精灵云两位联合创始人历时近一年的筹划终于面市，可说是大师之作。相比市面上已有的 Docker 书籍，本书在对存储驱动、Daemon 参数等方面的介绍都有不少值得圈点之处。相信阅读此书的读者能够在两位创始人手把手的指导下从零开始学会 Docker 的使用，为大规模运用容器技术做足准备。

—— 林帆，《CoreOS 实践之路》作者，ThoughtWorks 高级咨询师

Docker 是未来云计算的基础设施，是大数据、物联网的基础。正如 Docker 的口号“Build, Ship, and Run Any App, Anywhere”，Docker 以轻量、敏捷、灵活、经济等特点迅速赢得了开发人员和运维人员的青睐，容器技术也成为 DevOps 的首选。目前 Docker 发展势头迅猛，加之国内外相关资料太少，这本书的出现无疑是 Docker 学习者的福音。该书基于全新的 Docker 版本，辅以大量的操作实例，相信一定能让读者“开卷有益，手不释卷”。

—— 范东来，《Hadoop 海量数据处理》作者，《解读 NoSQL》《NoSQL 权威指南》译者，BBD（数联铭品）大数据技术部负责人

本书深入浅出、系统全面地介绍了 Docker 的应用，既可以作为快速入门，也可以作为操作手册，常备案头。精灵云在这个领域一直深耕细作，积累了丰富的实操经验，相信大家一定能从本书中学到干货。

程序设计是一门技术，还是一门艺术，技术是因为逻辑严密，艺术则是因为架构的美妙，使用 Docker，令程序解耦，就是艺术。HOMYi 在生产环境已经全面使用 Docker，解决突发大并发的问题已经变得更为容易，快速生长部署及程序发布也已经被大大简化。相信买了本书的你一定不会失望，窥探虚拟化云计算下程序架构的美妙逻辑。

—— 朱珠，步速者科技 CEO

上架建议：计算机>Docker

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍
责任编辑：张　慧
封面设计：吴海燕

ISBN 978-7-121-30244-2

定价：55.00元